12-2020

Nature-Inspired Topology Optimization of Recurrent Neural Networks

AbdElRahman A. ElSaid
aae8800@rit.edu

Nature-Inspired Topology Optimization of Recurrent Neural Networks

by

AbdElRahman A. ElSaid

A dissertation submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
in Computing and Information Sciences

B. Thomas Golisano College of Computing and
Information Sciences

Rochester Institute of Technology
Rochester, New York
December 2020

By

Ahmed Elsaid, AbdElRahman Ahmed

Committee Approval:
We, the undersigned committee members, certify that we have advised and/or supervised the candidate on the work described in this dissertation. We further certify that we have reviewed the dissertation manuscript and approve it in partial fulfillment of the requirements of the degree of Doctor of Philosophy in Computing and Information Sciences.

Dr. Travis Desell — Date
Dissertation Advisor

Dr. Alex Ororbia — Date
Dissertation Committee Member

Dr. Daniel Krutz — Date
Dissertation Committee Member

Dr. Risto Miikkulainen (UTexas) — Date
Dissertation Committee Member

Dr. Jan Van Aardt — Date
Dissertation Defense Chairperson

Certified by:

Dr. Pengcheng Shi — Date
Ph.D. Program Director, Computing and Information Sciences

Nature-Inspired Topology Optimization of Recurrent Neural Networks

by

AbdElRahman A. ElSaid

Submitted to the
B. Thomas Golisano College of Computing and Information Sciences Ph.D.
Program in Computing and Information Sciences
in partial fulfillment of the requirements for the
Doctor of Philosophy Degree
at the Rochester Institute of Technology

Abstract

Hand-crafting effective and efficient structures for recurrent neural networks (RNNs) is a difficult, expensive, and time-consuming process. To address this challenge, this work presents three nature-inspired (NI) algorithms for neural architecture search (NAS), expanding on the recent subfield of nature-inspired neural architecture search (NI-NAS). These algorithms, based on ant colony optimization (ACO), progress from memory cell structure optimization, to bounded discrete-space architecture optimization, and finally to unbounded continuous-space architecture optimization. These methods were applied to real-world data sets representing challenging engineering problems, such as data from a coal-fired power plant, wind-turbine power generators, and aircraft flight data recorder (FDR) data.

Initial work utilized ACO to select optimal connections inside recurrent long short-term memory (LSTM) cell structures. Viewing each LSTM cell as a graph, simulated ants would choose potential input and output connections based on the pheromones previously laid down over those connections as done in a standard ACO search. However, this approach did not optimize the overall network topology of the RNN.

This limitation was addressed by development of the Ant-based Neural Topology Search (ANTS) algorithm to directly optimize entire RNN topologies. ANTS utilizes a discrete-space superstructure representing a completely connected RNN where each node is connected to every other node, forming an extremely dense mesh of edges and recurrent edges. ANTS additionally selects node types from a library of modern RNN memory cells. Simulated ants then select paths and node types which are used to build

RNNs from the superstructure determined by pheromones laid out on the superstructure's connections. Backpropagation is then used to train the generated RNNs in an asynchronous parallel computing design to accelerate the optimization process. The pheromone update depends on the evaluation of the tested RNN against a population of best performing RNNs. Several variations of the core algorithm were investigated to evaluate heuristics involving different functions that drive the underlying pheromone simulation process as well as by introducing simulated ants with 3 specialized roles (inspired by the specialization of real-world ants) to construct RNN structures. This characterization of the agents enables ants to focus on specific structure building roles. "Communal intelligence" was also incorporated, where the best set of weights was across locally-trained RNN candidates were re-used for weight initialization, reducing the number of backpropagation epochs required to train each candidate RNN and speeding up the overall search process. However, the size of the superstructure in this approach needs to be specified by the user and also scales exponentially in terms of the number of its nodes, proving to be a major limitation.

To address this, the continuous ANTS (CANTS) algorithm was developed, which utilizes a continuous search space to indirectly encode network topologies. In this process, continuous ants (or *cants*) move within a temporally-arranged set of continuous/real-valued planes based on proximity and density of pheromone placements. The movement of cants over these continuous planes, in a sense, more closely mimicks how actual ants move in the real world. This continuous search space frees the ant agents from the limitations imposed by ANTS' discrete massively connected superstructure, making the structural options unbounded when mapping the movements of ants through the 3D continuous space to a neural architecture graph.

The three applied strategies yielded three important successes. Applying ACO on optimizing LSTMs yielded a 1.34% performance enhancement and more than 55% sparser structures (which is useful for speeding up inference). ANTS outperformed the NAS benchmark, NEAT, and the NAS state-of-the-art algorithm, EXAMM. CANTS showed competitive results to EXAMM and competed with ANTS while offering sparser structures, offering a promising path forward for optimizing (temporal) neural models with nature-inspired metaheuristics based the metaphor of ants. Further, CANTS accomplished this while requiring only half the number of user-tuned hyperparameters as ANTS and EXAMM.

Acknowledgments

Dr. Travis Desell has been my advisor and mentor throughout my post graduate journey at two different universities and in three different programs. I learned from him most of my academic, and research related knowledge, and much of my professional and communication skills. He has always been supportive and encouraging, leading me through my research path and teaching me how to fish on my own.

Dr. Alexander Ororbia II, my co-advisor, was an endless source of new ideas that opened my eyes to new perspectives on my work and the domain of my thesis. I learned much from his research approach and his speciality in nature-inspired machine learning methods.

I am thankful to Dr. Risto Miikkulainen, for honoring me as my external-advisory-committee-member and accepting to advise me through this work. His research has always been a reference to my work.

I owe so much to my colleagues in both the Distributed Nature Inspired Systems Lab (D2S2), under Dr. Desell, and Neural Adaptive Computing (NAC), under Dr. Ororbia, at RIT. They have been the best aid when things got hard. It has been always a joy to discuss my work with them, listen to their ideas, and collaborate with them. Thanks to Zimeng Lyu, and Joshua Karns.

Last but least, I acknowledge the hard work my professors made to teach me and shape my academic and research cognition.

This material is in part supported by the U.S. Department of Energy, Office of Science, Office of Advanced Combustion Systems under Award Number #FE0031547 and by the Federal Aviation Administration and MITRE Corporation under the National General Aviation Flight Information Database (NGAFID[1]) award. I also thank Microbeam Technologies, Inc. for their help in collecting and preparing the coal-fired power plant dataset. Most of the computation of this research was done on the high performance computing clusters of the Research Computing at Rochester Institute of Technology and the Data Center high performance computing clusters at the University of North Dakota. I would like to thank the Research Computing team at RIT and Mr. Aaron Bergstrom at UND for their assistance and the support they generously offered to ensure that the heavy computation this study required was available.

[1] https://ngafid.org

To my parents, to my family, and especially to my life-partner whom this work could not have seen the light without their support...

To my teacher and mentor...

Thank you!

Acronyms

ABC	Artificial Bee Colony
ACO	Ant Colony Optimization
ANN	Artificial Neural Network
ANTS	Ant-based Neural Topology Search
AR	AutoRegressive
ARIMA	AutoRegressive Integrated Moving Average
BP	Backpropagation
BPTT	Backpropagation Through Time
CANTS	Continuous Ant-based Neural Topology Search
CNN	Convolutional Neural Network
DAG	Directed Acyclic Graph
DNN	Deep Neural Network
GRU	Gated Recurrent Unit
ENAS	Efficient Neural Architectural Search
EXAMM	Evolutionary eXploration of Augmenting Memory Models
FDR	Flight Data Recorder
LSTM	Long Short Term Memory
MA	Moving Average
MAE	Mean Absolute Error
MGU	Minimum Gated Unit
MPI	Message Passing Interface
MSE	Mean Squared Error
NARX	Nonlinear AutoRegression with eXogenous inputs
NAS	Neural Architecture Search
NBJ	Nonlinear Box-Jenkins
NE	Neuro-evolution
NEAT	NeuroEvolution of Augmenting Topologies
NOE	Nonlinear Output Error
PSO	Particle Swarm Optimization
RNN	Recurrent Neural Network
SI	Swarm Intelligence
UGRNN	Update-Gated RNN
VAR	Vector AutoRegressive

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The problem of determining an optimal neural network architecture for a given problem, also known as neural architecture search (NAS) or neuroevolution (NE), is still an open question. This thesis investigates new strategies for NAS/NE through novel nature-inspired (NI) algorithms based on ant colony optimization (ACO), providing significant expansion to the new field of nature-inspired neural architecture search (NI-NAS). The strategies are examined on applied real-world problems, with the motivation to explore the capabilities, quality, and power of machine learning based on deep artificial neural networks (ANNs), to provide affordable and practical solutions for the industrial and technological challenges of today. As the potential search space for neural networks is unbounded, heuristic machine learning approaches can provide potentially effective and efficient solutions to these problems. This thesis contributes a deeper understanding of the technical and topological characteristics of neural network models as well as advancements to the field of ant colony optimization and nature-inspired algorithms.

1.1 Challenges in Neural Architecture Search

When attempting to solve a problem using an artificial neural network (ANN), the structure for the ANN is typically chosen based on its reputation given previous use in literature in the machine learning community related to solving similar problems. However, changing even a few problem-specific parameters can lead to poor generaliza-

tion when using a specific topology [3,47,47]. Traditionally, ANN structures are altered manually to achieve better performance, which often does not yield topologies that are near optimal with respect to desired ANN metrics such as complexity, performance, and accuracy. Realizing this requires an effective optimization technique to cover the vast search space, comprising of factors influencing the size, structure, occurrence of saddle points, and eventually the performance of the ANN itself. The multiple factors influencing this optimization problem, such as the non-convex nature of the problem, the occurrence of local optima, and the non-linearity of most of the applications which ANNs are proposed to solve, pose serious challenges for analytical optimization techniques [46,59,70,94,124]. To better determine what ANN to use, different architectures should be robustly explored and examined to find the ideal solution for a specific problem and/or use case.

Manually optimizing ANN structures has been an obstacle to the advancement of machine learning as it consumes significant time and requires a considerable level of experience in the domain [167]. Due to the ever increasing sizes of ANN models designed to meet the demand of the ever increasing complexity of data science and machine learning problems, neural architectural search (NAS) and neuro-evolution (NE) became a logical next step towards the automation of ANN design [46]. However many NE strategies rely on constructive methods that build the architectures from basic structural elements [156] and many NAS methods only operate within a highly limited search space [52].

The potential search space for neural architectures is potentially unbounded making the search problem extremely challenging. Further, determining an optimal architecture involves use specific trade-offs between accuracy and throughput. On the one hand, some use cases may want to trade some of the output accuracy to obtain fast results or to run the algorithms on limited computing and memory resources (*e.g.*, in the case of resource limitations on hand-held or embedded devices). On the other hand, other applications require highly accurate results with less limitations based on runtime or resources. Other optimization criteria can include efficient energy consumption, especially for systems designed for long running production use. The automation of designing an ANN structure can potentially adjust/affect the optimization function and criteria based on the problem the network is being designed for, something which normally would take a significant time for a human designer.

In addition, NAS can be prohibitively expensive, especially as it may require evaluating thousands or even millions of networks, each of which may require training and evaluation. The computation time and power required is considerably high when back-propagation of errors [127] (backprop) is used for training large-scale modern networks [158] – the de-facto method for training an ANN [5]. These problems are worse in the case of recurrent neural networks (RNNs) where stochastic gradient descent used in the backprop through time (BPTT) [102, 126, 153] algorithm, where training process is even more challenging than that for standard feed forward networks, incurring a noisier search terrain associated with the vanishing and exploding gradient problems [63,64,135]. Designing a procedure for optimizing network structures can alleviate the noise coming from elements which do not contribute positively to the solution by removing them from the architecture [80], making the training process easier by crafting sparser RNNs [151].

This study investigates nature-inspired NAS (NI-NAS), which applies nature inspired algorithms to the automated design of ANNs, as opposed to NAS, which typically uses gradient based or reinforcement learning based strategies, or NE, which utilizes evolutionary algorithm based strategies. The algorithms I developed are based on ant colony optimization (ACO) which utilizes simulated ants that traverse paths based on previously deposited pheromone values to guide the search. ACO was considered because the original problem that it was introduced for – with impressive results – were graph optimization problems [7, 30, 31, 33, 34, 96, 98, 115], and the structure of neural networks themselves were directed graphs. The similarity between the two problems and the success of ACO strongly motivated the work underlying this thesis.

The first two algorithms developed and investigated in this thesis are based on traditional ACO for the optimization process. These either 1) use a smaller predefined search spaces of a recurrent neural network (RNN) memory cell using an algorithm called *memory cell ant colony optimization* (MC-ACO) or, 2) use a massive search space of all potential architectures for a model called *ant-based neural topology search* (ANTS), which samples the candidate structures from discrete search spaces seeking an optimal model, as is done in traditional ACO. These approaches share similarity to NAS methods in ANN cell and architecture design [15,90,116,158], where candidates are generated from a bounded search space. The third algorithm I am proposing is a novel approach to both ACO and NAS, where ants make moves across a continuous search space, i.e., *continuous ANTS* (CANTS), allowing for any number of nodes and edges,

where the movements and stopping points of each ant are mapped to a discrete network architecture. This approach both removes the problem of a graph-based search space with pre-defined bounds and also more closely mimics how ants move in the real world. It also does this with fewer tune-able hyper-parameters than the prior approaches.

This dissertation contributes a deeper understanding of the technical and topological characteristics of neural models, validating results by designing RNNs that can perform time series data prediction on challenging real-world engineering problems. Furthermore, this work fundamentally expands the growing research domain of nature-inspired neural architecture search (NI-NAS) by presenting nature-inspired algorithms based on ACO for the design of recurrent networks. CANTS, in particular, provides a novel, powerful framework for both ACO and NAS, allowing ant agents to move within an unbounded, continuous search space to create network topologies without predefined limits on the number of nodes and edges.

1.2 Ant Colony Optimization for Neural Architecture Search

Most of the NAS methods used today operate within a limited search space, generating cells or architectures with a pre-determined number of nodes and edges [46], while most NE methods are constructive (e.g. NEAT [140]); they start with a minimum configuration for an ANN and then add and/or remove elements through the iterations of the evolution process. These methods apply genetic algorithms to evolve the neural structures through speciation, crossover, and mutation. Even more advanced strategies which involve generative encoding (a strategy to generate the network architecture and assign weights), such as HyperNEAT [139], require manually specifying the size of the generated architecture in terms of the number of layers and nodes. In contrast, Ant Colony Optimization (ACO) was introduced as a technique for a graph optimization problems [98], where the structure of an optimal subgraph needs to be found. ANNs are in their essence directed graphs which make them a similar problem to the one that ACO was originally designed for.

The first ACO algorithm (the "Ant System" [34]) was designed for the traveling salesman problem but failed to produce competitive results. However, subsequent research has shown that this algorithm can be effective on this problem [7, 33, 98] and interest in ACO has inspired many algorithms, including variants developed for continuous opti-

mization problems [8,36,76,77,103,133], and even implementations focused on training ANNs [10,69,97,120].

ACO has since been expanded to be used as a metaheuristic for finding approximate solutions to many combinatorial problems [30], earning its place in the vast family of nature-inspired algorithms [51,163]. ACO is also considered to be a swarm intelligence (SI) method, exhibiting many appealing properties such as fault tolerance, decentralisation, scalability, and the ability to share/combine the knowledge of swarm agents, which is extremely useful for NAS problems [14]. Moreover, the decentralization of ACO makes it a perfect candidate for a parallel programming design to utilize multiple computing units, rapidly accelerating the process. ACO employs a distributed approach using agents called artificial ants. These artificial ants resemble real ants, in that each ant is independent and communicates with other members of the colony through hormonal chemicals called pheromones. Ants randomly explore areas in their environment; however, they tend to follow paths with traces of pheromone left by ants who previously traversed the routes. Upon finding food they mark their return path with additional pheromone secretions. Pheromones decay over time, and paths with the highest pheromone levels represent the most promising routes leading to food (or, in other words, a "goal"). In this thesis, I designed my optimization process based on this ant colony/system behavior and draw more heavily and readily from current knowledge as to how ants behave.

Despite ACO's significant achievements, which are reflected in the literature of the heuristic optimization community, and although it is easily applied to directed graphs, the method has rarely been applied to neural network topology optimization. The exploration of ACO capabilities to evolve neural architectures is mostly a novel idea in the NAS realm. Previously published works have utilized ACO for NE; however, they were limited to small structures based on Jordan and Elman RNNs [29] or to only reduce the number of network inputs [100], whereas this work applies it to RNNs with orders of magnitude more complexity.

The fact that ACO can start its optimization work from a massive search space and construct the structures generated from within that search space makes the problem method interesting to investigate its results and compare it to existing NAS and NE methods. The algorithms in this work generate neural network structures from a massive discrete or an unbounded continuous search space, rather than growing or selecting

them from a minimal structure as commonly done in NE and NAS. The synthetic ants build the RNNs using their basic constructive elements: nodes, edges, and recurrent edges. In addition, node types are considered in the optimization process where the ants decide the type of each node in the RNN. The ants choose from a collection of memory cells, giving each RNN node a recurrent memory. The collection of RNN memory cells include long short-term memory (LSTM) [128], gated recurrent units (GRU) [19], minimal gated units (MGU) [165], update gate recurrent neural network units (UGRNN) [22], and Δ-RNN [112] units. The large number of neuron-type options provide the optimization agents expand the search space even further, and offering new challenges and the potential for better performance.

In this thesis, three search spaces are constructed to compare the outcome of search process in each of them. In the first, ACO is utilized to optimize connections within different fixed LSTM architectures. The second, uses a superstructure of connections representing all possible edges and recurrent edges connecting each and every node to the others in all time lag steps in the entire network. The time lags are used to create the recurrent connection of the generated RNNs. This superstructure represents an enormously dense RNN, providing opportunities to sample substructures which can be very deep RNNs. Each connection in the superstructure maintains a level of pheromone value, which is used to let the colonies' agents decide which connection they should take as they move from input nodes to the output nodes. Pheromone values on a connection are increased if the connection was part of a neural network that gives better performance compared to previously known performances in the evolution process. The third is a continuous search space, where the ants swarm without any fixed graph structure to select their path and deposit their pheromone traces. The search space is divided across levels, with each representing a time lag from which recurrent connections will pass information between different time steps of the RNN. This implementation does not enforce any bounds on neural network size or structure, apart from the number of ants used to swarm through the space. The complete details of these implementations are discussed in Chapter 4.

1.3 Research Objectives and Contributions

This study approaches the NAS problem from a different perspective as compared to most NE methods, utilizing ant colony optimization and nature-inspired neural architecture search as a metaphor. Whereas most NE methods evolve their structures in a constructive fashion, the methods used in this thesis sample a massive search space or use an indirect encoding from an unbounded continuous search space to converge to near to optimal solutions. Results show that the developed ACO approaches are very competitive with the existing constructive NE methods for recurrent networks and, notably, can require significantly fewer hyper-parameters to hand-tune.

A Lamarckian-like *communal intelligence* weight sharing mechanism was designed and implemented to investigate the effect of reusing the trained weights of the evaluated neural networks on convergence of the NAS methods used, yielding a powerful initialization scheme for RNN candidates created and tuned locally by the ACO algorithms. The results reveal that this communal weight sharing scheme strongly aids in the convergence of the optimization process. Additionally, unconventional L1 and L2 based pheromone regularization techniques were applied to the pheromone deposition of ANTS, with the goal of encouraging the colony to build sparser structures. The structures optimized using this technique were among the top performing structure in the results of the study. Other measures were also experimented to test the effect of these regularization techniques on the sparsity of the structures and their performance. The movement of the ants was one of them. Those measures also proved successful in giving the ants the freedom to skip layers in the discrete search space. Furthermore, utilizing different species of ants by giving them specific roles in the construction of the neural networks provided further benefits to the performance of the resulting structures.

The search spaces studied in this dissertation were first represented by a discrete search spaces and then by a continuous search space. The goal was to offer the optimization agents an unbounded search space to explore without requiring the user to place bounds on the number of layers nodes and edges. This approach also allows for more freedom in the design of the ant agents. This continuous approach showed competitive results to the other implementations while at the same time requiring less tunable hyperparameters, making it easier to use.

1.4 The Organization of the Dissertation

The rest of this dissertation is organized as follows: Chapter 2 provides the basic background needed to understand the concepts developed throughout. Chapter 3 reviews the literature for studies that relate to the focus of this thesis and the its applied concepts. Chapter 4 introduces the methods and algorithms used in my experimental studies. Chapter 5 presents the results of the experiments performed on the implemented methods. Chapter 6 concludes the dissertation, summarizes the key results, and presents future directions for research.

Chapter 2

Time Series Data Prediction and Artificial Neural Networks

The applied problems used to evaluate the ACO methods developed in this thesis involve time-series data prediction from real-word data sets. Therefore, this chapter offers a general introduction to the basic concepts used in this work. The chapter describes time-series-data prediction and the methods typically associated with this machine learning problem: typically statistics-based and ANN based methods. The sections will also introduce the types of ANNs, deep neural networks, recurrent neural networks, and memory based cells commonly used for these problems.

The main goal of time series data prediction is to provide vital information for decision makers, economists, plans optimizers, industrialists and critical systems operators. There are two sides to prediction: qualitative and quantitative. Qualitative methods are classified as judgmental or subjective prediction methods, and includes methods relying on intuition, judgment, or opinions of some kind of referee like consumers, experts and/or supporting information. Qualitative methods are mainly applied in cases when past data is not available. Quantitative methods are used when previous information obtained from logged data is available. Quantitative methods can be either univariate or multivariate. For most real-world problems, the time series data consists of several dependent variables, and as a result for these problems multivariate prediction methods are used [16].

The models used for time series prediction realm mainly fall into three categories: *a)* statistical prediction models, *b)* artificial neural networks (ANNs), and *c)* hybrid methods. These methods are detailed as follows.

2.1 Statistical Prediction Models

These models include, but are not limited to, the autoregressive (AR) model, the moving average (MA) model, and hybrid models that derive from them such as autoregressive moving average (ARMA), autoregressive integrated moving average (ARIMA), seasonal ARMIA (SARIMA) [11], and vector autoregressive (VAR) models.

These models are less sophisticated compared to the ANN and hybrid models. Therefor, given the results of the experiments done on the hybrid and the ANN models (Section 5.2) and the fact that all the data used in this study belong to a similar complicated engineering systems, statistical models are considered beyond the scope of the study.

2.2 Artificial Neural Networks (ANN) Prediction Models

While statistical prediction models have managed to achieve good results and have even reported results that outperform neural networks [58], the main drawback of statistical methods is that they are generally inefficient to non-linear systems, whereas ANNs have shown good performance when applied to such systems [11]. ANNs are an imitation of human neuron cells as shown in Figure 2.2a, where neurons (nodes/perceptrons) take input data and perform simple operations then selectively pass the results onto other neurons [88]. McCulloch and Pitts developed the first artificial neuron model in 1943 [11]. However, ANNs emerged in the 1950s which was built using simple simulations of biological neurons called perceptrons. Perceptrons take several binary inputs and produce a single binary output. To compute the output, real numbers called weights are used to represent the importance of the respective inputs to the output as shown in Figure 2.2b. The perceptron's output, 0 or 1, is determined by whether the weighted sum $\sum_{j=1} w_j \cdot x_j$ is less than or greater than some threshold value.

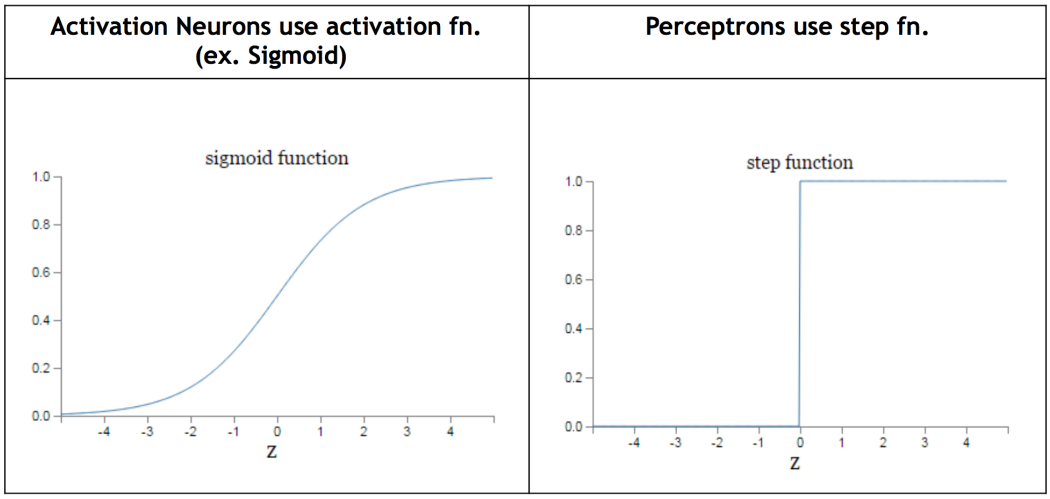
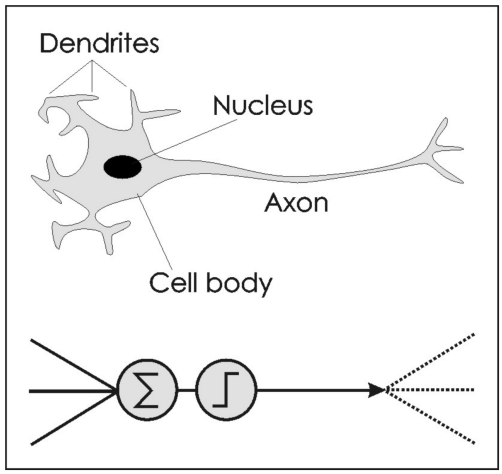


Figure 2.1: *Activation-Based Neurons Vs. Perceptron Neurons* [107]

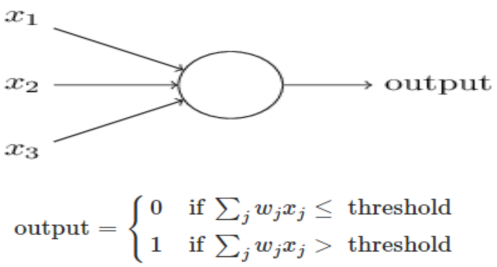
Layers of perceptrons are used where simple decisions are made in early layers and more complex and abstract decisions could be taken in later layers as shown in Figure 2.3a. Thus, perceptrons can be looked at as a method for weighing evidence to make decisions. However, a small change in one of the weights can completely flip the output, as the output is binary. This was resolved by introducing 'Activity-driven neurons', which apply an activation function to the values from incoming connections. The advantage of these types of neurons is that a small change in their weights causes only a small change in their output. The difference between a perceptron neuron and an activation neuron is shown in Figure 2.1.

2.2.1 ANN Training

There are several algorithms that can be used to set the weights of an ANN to represent a desired function. The most popular method is a supervised learning technique called backpropagation (BP). Backpropagation is done by first forward propagating the input data through the input and hidden layers, and then computing the predicted value(s) at the output layer. After that, the difference (error) between the predicted and expected output is calculated from a cost/loss function, which is derived from an objective function that represents the problem's constraints. The cost function has a global minima at a point at which the target value and network output are nearly equal. Next,

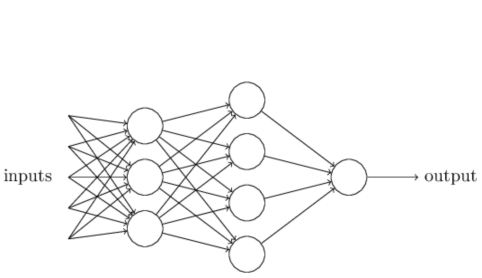


(a) *Biological - Artificial neurons* [107]

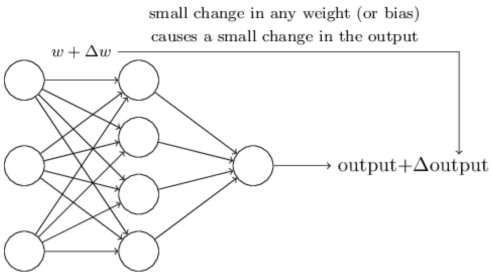


(b) *Perceptron input/output* [107]

Figure 2.2: *The Perceptron.*



(a) *A Feed Forward Neural Network* [107]



(b) *Learning Process* [107]

Figure 2.3: *Forward & Backward propagation of information.*

the error, obtained by the cost function, is fed backwards (backpropagated) through the network to modify the weights based on the gradient of the error with respect to weights. This is done repeatedly until the total error over the training data is minimized, converging weights of the ANN along with the average of the errors associated with each desired output. The neural network then extrapolates in order to classify (predict) vectors of input signals that it had not yet been presented with. This process is shown in Figure 2.3b [107].

While some studies like Desell *et al.* [29] (Particle Swarm Optimization), Ororbia *et al.* [110, 111] (Local Representation Alignment), Tang *et al.* [144] (genetic algorithm), and Igel [68] (covariance matrix of the mutation distribution), devised and used derivative-free optimization (DFO) algorithms to optimize RNN weights, the ability of DFO methods *"to obtain good solutions diminishes with increasing problem size"* [125].

2.2.2 ANN Weight Initialization

Weight initialization plays a big role in the success of the neural networks. A good weight initialization strategy can speed up the training process. A variety of weight initialization techniques, like Xavier [57] and Kaiming [62], were invented to address vanishing and exploding gradients and to speed the training process. Neuro-evolution can algorithms provide even more effective weight initialization strategies. In the process of neural network optimization, weights can be considered as an optimizable characteristic of the structure. In NEAT [140], where neural network structures are regarded as individuals in a living and mating population, weights (as part of the genome structure) are passed from a generation to the next in the matching parental chromosomes. However, this weight inheritance is done randomly in NEAT: an offspring can have a chromosome from either one of their parents regardless of who is more fit.

Real *et al.* [123], EXACT [27], Prellberg *et al.* [119], EXAMM [109], and LEMONADE [45] use a Lamarckian[1] weight inheritance strategy – also called an Epi-genetic [27] strategy. EXACT, LEMONADE, Real *et al.*, and Prellberg *et al.* apply the strategy

[1] A theory about biological evolution which states that organisms alter their behaviour in response to the change in their surrounding environment and those who fail to respond to the changes become extinct. Those changes are then passed to the organisms' offsprings genetically. Through this process, only successful changes necessary to cope with environmental changes are the ones that prevail.

to CNNs and EXAMM applies this to RNNs. EXAMM and EXACT apply a randomized line search to generate child parameters, offering more flexibility to exploring the search space. Using this strategy, EXAMM and EXACT were able to reuse weights from previously trained parents to initialize newly generated NNs to speed the optimization process by requiring fewer training epochs. The Prellberg *et al.* implementation was a simple procedure where only mutated components were randomly re-initialized.In the strategy presented by Real *et al.*, weights can be re-used for generated genomes and their mutations. Some mutations preserve all the weights of the parental structures, some have their weights reinitialized, and the majority preserve some of the weights and reset the others.

2.2.3 ANN Types

Many variations and types of ANNs have been developed. Differences between these variations and types could be in the architecture, choice of activation function, or learning algorithms, etc, depending on the proposed problem. The most traditional and common ANN is the feed-forward neural network (FFNN) or multilayer perceptron (MLP). Figure 2.3a presents a basic feed-forward neural network architecture [67]. Other more advanced ANN architectures, such as those based on recurrent neural networks (RNNs), include traditional RNNs like Jordan [72] and Elman [37] networks, as well as more complicated models that use memory cells (*e.g.*, long short-term memory (LSTM) cells [49,65,67]), or other novel designs such as echo state networks (ESN) [71]. Other ANNs architectures include radial basis function (RBF) [13], and cascading neural networks [56]. All these variations of ANN design can be applied to time series data predictions. The parameters of the data are fed to the ANN through its input nodes and data flow from the input nodes to the successive layers. Moving from a layer to the following, the information is abstracted and analyzed through the internal structure of the ANN, and the values obtained at the output nodes present the desired predictions/classification. However, only RNNs have structures capable of retaining information from previous time steps.

Deep Neural Networks (DNN)

Biological brain cells process information and pass the results to the consecutive cells through their synapses. This flow of information offers abstraction and therefore makes analysis and decision making easier [99, 127]. While for many years it was thought that the depth of a neural network would be restricted to at most two hidden layers due to limitations in training strategies, recent advancements in AI have been made by understanding more about training heuristics, *i.e.*, tackling vanishing and exploding gradients, and improvements in hardware, which have made it possible for NNs to add layers and grow deeper. This provides DNNs with more features' extraction and abstraction powers. Through these levels of abstraction, DNNs can compactly represent highly varying functions, which entail very large sized architectures [4].

Recurrent Neural Networks (RNN)

RNNs [127] offer past temporal information to current passes through the RNN architecture by connecting neurons with previously assessed information through recurrent connections. This temporal dynamic behavior characterizing RNNs give them memory, which assists the information abstraction performed by neurons by feeding information from the past time steps.

Memory Based RNNs

The concept of recurrency and providing ANNs with "memory" has been accomplished by equipping neurons internal recurrent connections which can be controlled locally in the training process. The internal structure inside these types of neurons provides them with gated memory capable of offering abstraction to the information flowing through the neurons. More importantly, the memory passes temporal information from past time series to current time series, offering higher level recurrency to the RNN and the potential to latch on to earlier information. Examples of these types of neurons are:

LSTM RNN LSTM RNNs were first introduced by Hochrieter & Schmidhuber [128]. LSTM neurons provide a solution for the exploding/vanishing gradients problem by uti-

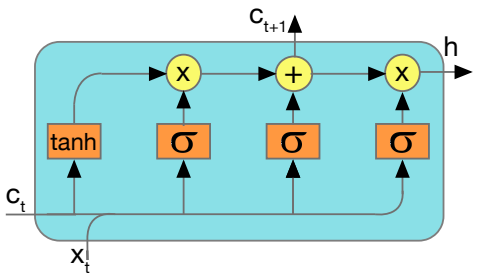


Figure 2.4: *The LSTM Cell.*

lizing various gates, which allow backpropagation to be used in large RNNs (S. Hochrieter in 1991). This work has paved the way for many interesting projects. Later, J. Schmidhuber *et al.* [49] emphasized the forget gate in the LSTM RNNs. However, Felix A. Gers *et al.* [54] suggest that *"LSTM's superiority does not carry over to certain simpler time series prediction tasks solvable by time window approaches"*. The paper suggests *"to use LSTM only when simpler traditional approaches fails"*.

Figure 2.4 illustrates the cell structure. The equations controlling the flow of data through the unit are as follows:

$$\begin{aligned} f &= \sigma(W_f * x + U_f * c_prev * f_{bias}) \\ i &= \sigma(W_i * x + U_i * c_prev * i_{bias}) \\ o &= \sigma(W_o * x + U_o * c_prev * o_{bias}) \\ c &= f * c_prev + i * tanh(W_c * x + c_{bias}) \\ h &= o * c \end{aligned} \tag{2.1}$$

where x is the cell input vector, h is the output of the cell, W_f is the forget-gate weights associated with the input, U_f is the cell's memory forget-gate weights, W_i is the input-gate weights associated with the input, U_i is the cell's memory input-gate weights, W_o is the output-gate weights associated with the input, and U_o is the cell's memory out-gate weights. There are 11 trainable parameters represented in the weights and biases of an LSTM cell.

Gated Recurrent Unit (GRU) GRU was introduced by Chung *et al.* [19] for the neural machine translation on long sentences of at most 30 words in English and French. The sentences were built using the 30,000 most frequent words in both languages. The

study used two models with different encoders. The language translation performance was good but did not perform well on relatively longer sentences.

Chung *et al.* [20] composed a study to compare LSTM to GRU on polyphonic music and speech data and found that GRU is actually comparable to LSTM and concluded that their "evaluation clearly demonstrated the superiority of the gated units".

Figure 2.5 illustrates the cell structure. The equations controlling the flow of data through the unit are as follows:

$$\begin{aligned} z &= \sigma(zw * x + zu * h_prev * z_{bias}) \\ r &= \sigma(rw * x + lu * h_prev * r_{bias}) \\ k &= hw * x + hu * r * h_prev * h_{bias} \\ h &= z * h_prev + (1 - z) * tanh(k) \end{aligned} \tag{2.2}$$

where x is the input vector, h is the output, rw is the reset gate input weight, lu is the reset gate previous output weights, r_{bias} is the reset gate bias, hw is the memory gate input weight, and hu is the memory gate previous output weights. There are 9 trainable parameters represented in the weights and biases of a GRU cell.

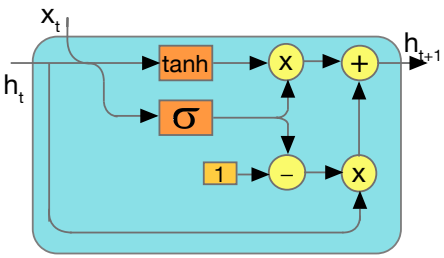


Figure 2.5: *The GRU Cell.*

Minimum Gated Unit (MGU) MGU was proposed by Zhou *et al.* [165] as a memory unit with only one gate. The unit was introduced as an alternative to LSTM and GRU with a "simpler structure, fewer parameters, and faster training".

Figure 2.6 illustrates the cell structure. The equations controlling the flow of data

through the unit are as follows:

$$\begin{aligned} f &= \sigma(fw * x + fu * h_prev * f_{bias}) \\ h &= tanh(hw * x + f * hu * h_prev * h_{bias}) \\ out &= (1 - f) * h_prev + f * h \end{aligned} \tag{2.3}$$

where x is the input vector, h is the output, f is the forget vector, fw is the forget gate input weights, fu is the forget gate previous output weights, hw is the output gate input weights, and hu is the output gate previous output weights There are 6 trainable parameters represented in the weights and biases of an MGU cell.

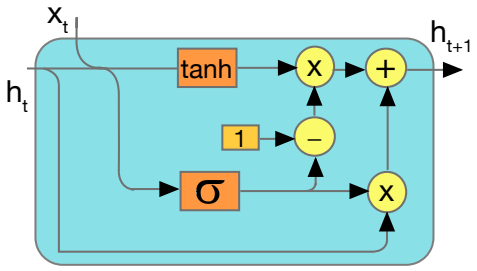


Figure 2.6: *The MGU Cell.*

Update-Gated RNN (UGRNN) Unit UGRNN was proposed by Collins *et al.* [22] as enhanced memory unit over the LSTM and GRU, where the authors contended the ease of training of networks using this type of neurons. Figure 2.7 illustrates the cell structure. The equations controlling the flow of data through the unit are as follows:

$$\begin{aligned} c &= tanh(cw * x + ch * h_prev * c_{bias}) \\ g &= \sigma(gw * x + gh * h_prev * g_{bias}) \\ h &= g * h_prev + (1 - g) * c \end{aligned} \tag{2.4}$$

where x is an input vector, h is the output. There are 6 trainable parameters represented in the weights and biases of an UGRNN cell.

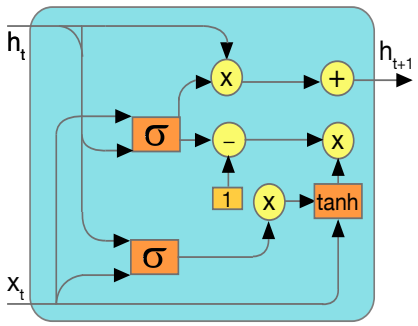


Figure 2.7: *The UGRNN Cell.*

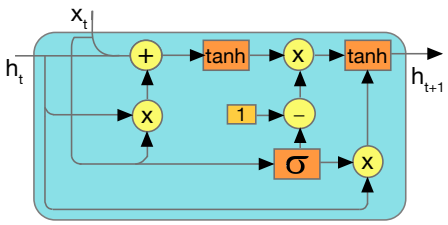


Figure 2.8: *The Δ-RNN Cell.*

Δ-RNN Unit Ororbia *et al.* [112] proposed the Δ-RNN unit in a study that involved language modeling. The unit performance was used to construct a novel RNN called the Δ-RNN which was evaluated on a series of popular text benchmarks. Figure 2.8 illustrates the cell structure. The equations controlling the flow of data through the unit are as follows:

$$\begin{aligned} \hat{h}_1 &= \alpha * (v * h_{prev}) * x \\ \hat{h}_2 &= \beta_1 * v * h_{prev} + \beta_2 * x \\ \bar{h} &= tanh(\hat{h}_1 + \hat{h}_2 + bias_{\hat{h}}) \\ r &= \sigma(x + bias_r) \\ h &= tanh\Big((1 - r) * \bar{h} + r * h_{prev}\Big) \end{aligned} \tag{2.5}$$

where x is the input, and h is the output. There are 6 trainable parameters resented in the weights and biases of a Δ-RNN cell.

2.3 Hybrid Models

These models offer the benefits of both statistical and ANN models, offering the simplicity of the statistical methods, potentially overcoming a long training process and

difficult to identify inputs, and the limited interpretability of the ANN models [152]. Ranković *et al.* [122] introduced a study which represents these models. Where Θ is the regressor, y is the target data, $\hat{y}$ is the output, e is the error, t is the time step, and n is the lag limit, the proposed neural networks in the study were the:

- *Nonlinear Finite Impulse Response (NFIR) model:*

$$\Theta(t) = (i_{t-1}, i_{t-2}, \ldots, i_{t-n})$$

- *Nonlinear AutoRegressive with eXogenous inputs (NARX) model:*

$$\Theta(t) = (i_{t-1}, i_{t-2}, \ldots, i_{t-n}, y_{t-1}, y_{t-2}, \ldots, y_{t-n})$$

- *Nonlinear AutoRegressive Moving Average with eXogenous inputs (NARMAX) model:*

$$\Theta(t) = (i_{t-1}, i_{t-2}, \ldots, i_{t-n}, y_{t-1}, y_{t-2}, \ldots, y_{t-n}, e_{t-1}, e_{t-2}, \ldots, e_{t-n}))$$

- *Nonlinear Output Error (NOE) model:*

$$\Theta(t) = (i_{t-1}, i_{t-2}, \ldots, i_{t-n}, \hat{y}_{t-1}, \hat{y}_{t-2}, \ldots, \hat{y}_{t-n})$$

- *Nonlinear Box-Jenkins (NBJ) model:* which uses all four regressor types.

NARX, NBJ, and NOE were considered as benchmark models to investigate their performance on highly non-linear and non-seasonal engineering data belonging to aircraft turbine engines and compare them the inital ACO-based method introduced. The results (discussed in Section 5.2) compared to the output of a both an unoptimized and ACO-optimized RNN with LSTM cells, which outperformed those hybrid models. The following describes the ANN formulations of these three hybrid models.

Nonlinear Output Error (NOE) Inputs Neural Network: The structure of the NOE network is depicted in Figure 2.9. The model has the inputs of several time lags fed to a hidden layer and the output of the hidden layer is fed to the model's output.

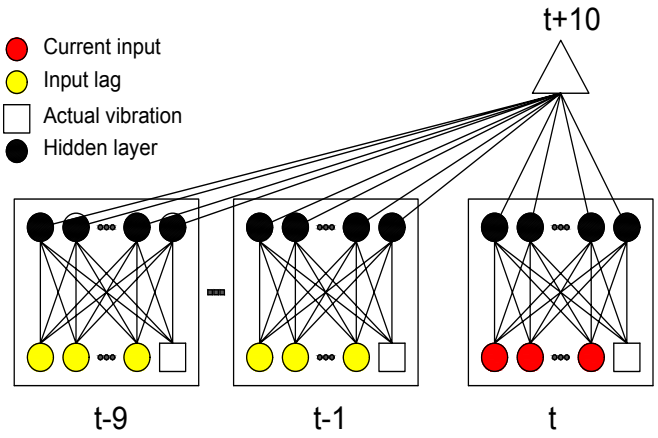


Figure 2.9: *Nonlinear Output Error inputs neural network. This network was expanded to utilize 10 seconds of input data.*

Nonlinear AutoRegression with eXogenous (NARX) Inputs Neural Network: The structure of the NOE network is depicted in Figure 2.9. This model is like the NOE model except that it also has the output at previous time steps fed as an input, which feeds the errors of those outputs to the network so it can learn from them as inputs.

Nonlinear Box-Jenkins (NBJ) Inputs Neural Network: The structure of the NBJ is depicted in Figure 2.11. This model is similar to the NARX model except that, in addition to the previous outputs, the error is calculated from the difference between the target value and the output is fed-in at previous time steps as input, giving the model stronger capability to learn from the error of the previously predicted outputs.

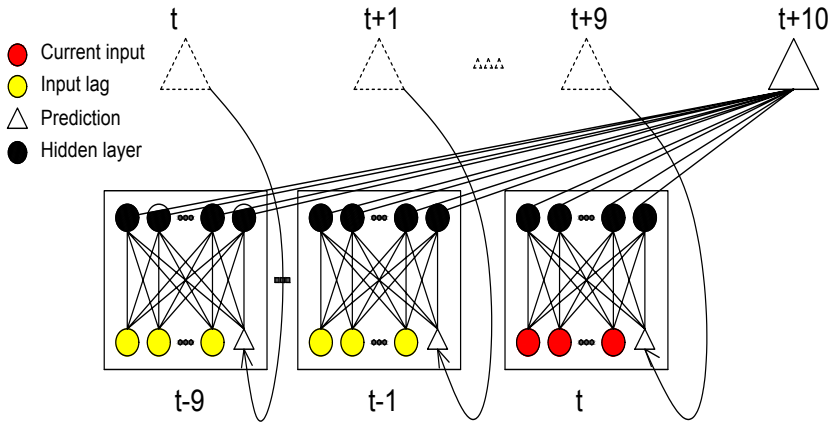


Figure 2.10: *Nonlinear AutoRegressive with eXogenous inputs neural network. This network was expanded to utilize 10 seconds of input data, along with the previous 10 predicted output values.*

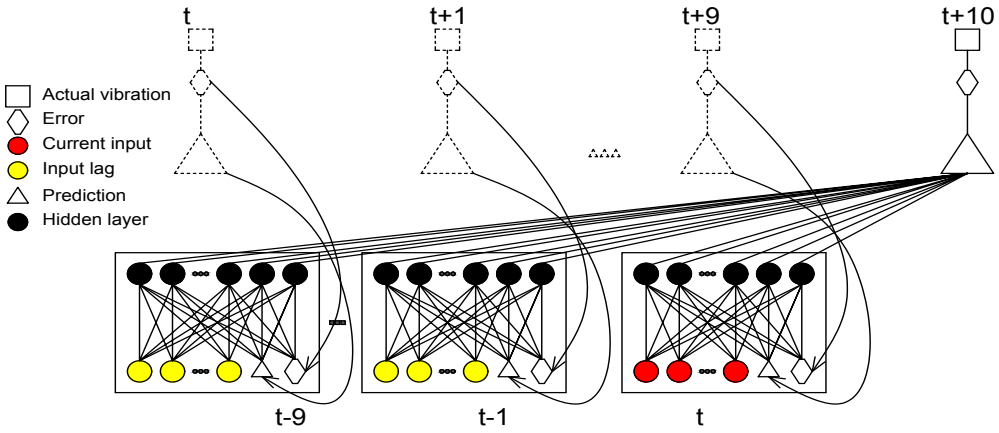


Figure 2.11: *Nonlinear Box-Jenkins inputs neural network. This network was expanded to utilize 10 seconds of input data, along with the future output and error values. Due to requiring future knowledge, it is not possible to utilize this network in an online fashion.*

Chapter 3

Related Work

The chapter first reviews studies involving time-series data prediction, in particular the studies related to the real world data domains used in this work (aviation, coal power plants and wind turbines) are discussed. Following this, it gives an overview of work in neural architecture search, neuroevolution, ant colony optimization, and other nature-inspired methods.

3.1 Industrial Time Series Data Prediction

Time series data plays a vital role in industrial system monitoring and reliability studies. This study utilizes data from three of these industries: aviation, coal-fired power plants, and wind power generation.

3.1.1 Time Series Prediction in Aviation

With respect to detecting engine abnormalities, *e.g.*, Nairac *et al.* [104] have proposed an ANN-based scheme trained to detect abnormalities in engine vibrations based on flight data recorder (FDR) data. To achieve good performance, the work used two modules. One of the modules uses the overall shape of the vibration curve to detect unusual vibration signatures. The second one reports sudden unexpected transitions in the signature curves. Their approach to detect defects is not to introduce examples of

faulty engines to the ANN, rather, only examples of healthy engines are introduced to the neural networks in the training phase. This approach was taken to overcome the lack of adequate training data for faulty engines. In this context, Nairac's paper introduces the term 'normality' to describe the behavior of normal engines and 'abnormality' to describe the behavior of faulty engines. By using statistical models, the faulty engines detection would be described as 'novelty' detection based on deviation from the data distribution. A prediction accuracy of 84% was successfully achieved.

David A. Clifton *et al.* [21] presented work for predicting abnormalities in engine vibration based on a statistical analysis of vibration signatures. The study presents two modes of prediction. One is ground-based (offline), where prediction is done via run-by-run analysis to predict abnormalities based on previous engine runs. The success in this approach was the prediction of abnormalities up to two flights ahead. The other model is a flight based-mode (online) in which detection is done either by sending reduced data to the ground-base or by processing it onboard the aircraft. The researchers argued that they could successfully predict vibration events 2.5 hours in the future. However, this prediction is made after half an hour of flight data, which might pose problems since excess vibration could occur during the data collection interval. The paper did not mention how much data were required to ensure good prediction.

Unlike FFNNs, RNNs can deal with sequential input data, using their internal memory to process sequences of inputs and use previously stored information to aid in future predictions. This is done by feedback connections (or introducing loops between neurons), allowing them to handle sequential data [55].

Part of this thesis research on RNN topology optimization to predict aircraft engine vibrations (published in [39]) was in part inspired by two studies on predicting flight parameters [28, 29]. The first study began with evolutionary algorithms such as particle swarm optimization [81, 117] and differential evolution [141], to optimize the weights of the network [28]. A later study used an ACO [9, 32, 35] based algorithm to evolve different RNN structures. The neural networks evolved with ACO predicted airspeed, altitude, and pitch achieving a 63%, 97% and 120% improvement respectively over the previous best published results.

3.1.2 Time Series Prediction in Energy Generation From Coal

Performance and emissions monitoring in electricity generating coal power plants are of vital importance for economy and environment aspects [147, 150, 161]. Therefore, there has always been a need to find models that give operators and engineers the ability to acquire knowledge about the future performance of the power plants to act proactively to increase productivity and alleviate chances of unfortunate outcomes. Analytical mathematical models were built to address this issue; however, those models are considered complex and time consuming [25].

Regression models were sought to use the actual operation history data to generate feasible prediction models. Li *et al.* [89] compared the prediction performance of grey-box, ARX, and NARX models to detect nitrogen oxides (NO_x) emissions. Lv *et al.* [95] used least squares support vector machine to predict NO_x.

Artificial neural networks were also present in predicting vital parameter in the industry of coal fired power plants. Yao *et al.* [161] studied the prediction of the hydrogen content of coal used in power station boilers. Azid *et al.* [2] used ANN to predict coal pollutant emissions. Tunckaya *et al.* [147] took a transition step made a comparison between applying regression models and ANNs in the problem of productivity in the coal power-generation industry. The study concluded that the ANN model made better results compared to ARIMA and Multiple Linear Regression (MLR) models.

Jorjani *et al.* used ANNs to predict both organic and inorganic sulfur removal from coal using a mixed culture consisting of *A. ferrooxidans* and *Leptospirillum ferrooxidans* species extracted from coal washery tailings, for pyritic sulfur, and *Rhodococcus P32C1* species, extracted from oily soils, for the organic sulfur [73]. They have also used ANNs to predict the effects of operational parameters on the organic and inorganic sulfur removal from coal by sodium butoxide [74]. The study reported good prediction results with correlations of $R_2 = 1.00$ and 0.98 for pyritic sulfur removal and $R2 = 1.00$ and 0.97 for organic sulfur removal in the training and testing stages. Teruel *et al.* use ANNs to predict ash deposits in coal-fired boilers, having developed their model with the aid of case study where a furnace was fouled as detected by heat flux meters [145]. The study used three networks to predict three parameters (effective blow, cleanliness, and evolution of heat flux) which indicate changes in the heat flux. Results with more than $R^2 = 0.9$ were achieved for the models when trained and tested on a dataset with

tens of thousands of records. De *et al.* have trained an ANN using existing plant data to model the biomass and coal cofired CHP plant of Västhamnsverket at Helsingborg, Sweden [25]. Cheng *et al.* have used ANNs to predict the maximum burning rate and fixed carbon burnout efficiency of 16 typical Chinese coals and 48 of their blends with a relative mean errors of 1.97% and 0.91%, respectively [17], as well as the ignition temperature and activation energy at 1.22% and 3.89%, respectively, in another work [18]. Liu *et al.* have used two different ANN structures to model a 1000 MW ultra supercritical once-through boiler unit of a power plant, showing efficiencies over a standard recursive least squares method [91]. Smrekar *et al.* have used two integrated ANNs, representing a turbine and boiler, to predict the power output of a coal fired power plant. The models were trained and validated on 745 rows of real plant data (65% training, 15% validation, and 25% testing). The best performing used model consisted of 9 neurons and was trained for 4,000 epochs. The research reported acceptable predictions rates [132]. Kumari *et al.* have predicted the fireside corrosion rate of superheater tubes in a coal-fire boiler using an ANN trained with operational data from an Indian thermal power plant [84]. The study used dataset of 1,000 records (50% training, 25% validation, and 25% testing). The experimented structures of the neural network comprised 9 input nodes, two and three hidden layers and one output node and outperformed statistical regression models. Zhou *et al.* utilized ANNs to predict the nitrogen oxides (NOx) emission characteristics of a large capacity pulverized coal fired boiler, showing a more convenient and direct approach compared to other modeling techniques, such as computational fluid dynamics [166]. Recently, Raman and Klima utilized an ANN to perform sensitivity analysis which indicated filtration time and pH were the most significant variables influencing filtrate flux when evaluating pressure filtration of coal refuse slurries [121]. The study conducted experiments where 82 feed forward ANN models were trained and evaluated. The model had R_2 value over 0.9 on both training and testing datasets, indicating the goodness of fit. Onat *et al.* have used ANN system to predict excess air coefficient (λ) on coal burners equipped with a CCD camera, and reached $R = 0.984$ in terms of accuracy compared to less than 13% for regression methods [108]. The study did not mention the structure of the used ANN but reported that it was trained using the Levenberg-Marquardt algorithm. Time-series based forecasting was done by Laubscher [86] using encoder-decoder recurrent neural networks. The model predicted reheater metal temperatures using stacked encoder and decoder sections with GRU units and 512 hidden units per layer, reporting an MAE below 1%. Also Tan *et al.* [143] used LSTM RNN on time-series data to predict NOX emission

in a 660 MW coal-fired boiler. MAE's are within 3%, outperforming results obtained using SVM. The study used a real power plant dataset of 10000 records (7 days of operation). The study experimented on 10 time lags, with 8 different number of hidden nodes (starting with 8 and ending at 1024 with multiples of 8). The best performing structure was the 256 hidden nodes one. Finally, Liu *et al.*approached the problem from an evolutionary perspective. They used ANNs trained with Ant Colony Optimization based Back Propagation (ACO-BP), instead of gradient-based training, to model coal ash fusion temperature based on its chemical composition [93]. The study utilized a 3 layer network with 10 hidden nodes. The oxide compounds were fed as inputs and the out was the fusion temperature. The data used belonged to Chinese coal power plant and consists of 80 ash samples. ACO-BP showed that it can obtain better performance compared unoptimized structures.

ANN are widely used in the industry starting from simple feed-forward ANNs, and expanding to using more complex RNN structures which are manually optimized, to using swarm intelligence (SI) to optimize the ANN parameters. Therefore, automating the optimization of ANN structures used in this industry is a natural step. Although, ACO is used in optimizing the weights of the ANN's [93], review of the literature did not reveal that it was used to optimize ANNs in problems related to this industry.

3.1.3 ANN for Predicting Wind Energy Generation Performance

Wind energy power generation farms produce abundant time-series data which can be used in predicting the industry operation parameters. This is a sample of applying ANN in the wind power generation industry:

Senjyu *et al.* [130] proposed an LSTM UGRNN trained on time-series data to forecasting wind speed and to perform wind power generation predictions. The study used Elman [37] type ANN and obtained results with MAE equal to 4.87% for 3 hours wind speed forecasting, and MAE of 14.15% for predicting the generated power 3 hours in future. Srivastava *et al.* [137] used RNN, GBM, and LSTM to predict the power generated from wind energy using wind velocity via wind turbine. The dataset used was annual hourly data collected in 2014 from a wind farm in Kolkata, India. They had only two parameters: wind velocity and turbine Power output, and it was divided to 70% training data and 30% testing data. The results showed that the LSTM model

outperformed the RNN and GBM models with MSEs equal to 17.67%, 26.54%, and 32.02% respectively. Sandhu *et al.* [129] composed a comparative study of a time series model using ARIMA and RNN with LSTM. The study involved 2 datasets and concluded that the LSTM RNN achieved an MAE with an 18.18% improvement over the ARIMA model. The study did not give details about the structure of the used LSTM RNN but the data used were two datasets of 129 and 865 hourly-based real-data points collected by the University of Massachusetts. Pradhan *et al.* [118] introduced a hybrid model consisting of two phases: *a)* decomposition of wind speed sample data by wavelet technique, and b̂) the utilization of these decomposed data to predict wind speed through recurrent wavelet neural network (RWNN). The study used real data to train and the validation of the model showed learning ability for the RWNN compared to the RNN without using wavelet decomposed wind speed.

3.2 Neural Architecture Search Methods

Neural Architecture Search (NAS), defined as "the process of automating architecture engineering, is thus a logical next step in automating machine learning" [46]. The goal of NAS can be accomplished through a variety of strategies. Many of the methods used in NAS are Evolutionary Algorithms (EA), and Reinforcement Learning [138]; therefore, Neuroevolution – which evolves neural networks using EA's – can be considered a subcategory of NAS. Commonly, genetic algorithms are most popular in NE [24,46,138], even though there are other algorithms which are used in NE like Monte Carlo-based simulations [105], random search [6] and random search with weights prediction [12], hill-climbing [44], grid search [162], and Bayesian optimisation [78,78]. However Swarm Intelligence (SI) algorithms like Particle Swarm Optimization (PSO) [50, 142, 148, 149] and nature-inspired (NI) algorithms such as the Artificial Bee Colony Optimization (ABC) [66], the Bat algorithm [160], the Firefly algorithm [159], and Cuckoo [87], are also being explored for NAS.

3.2.1 Reinforcement Learning Methods

Many NAS approaches utilize a controller neural network which learns how to optimize the hyperparameters of the topology. This idea was implemented by Zoph and Le [167]

in the Neural Architecture Search (NAS). Following the same concept but confining the NAS's search space in a directed acyclic graph (DAG), Pham *et al.* [116] introduced Efficient Neural Architectural Search (ENAS). While the topologies of the architectures in NAS are fixed as a binary tree and a controller only learns the operations at each node of the tree, ENAS encodes both the structure and the operations in the search space represented by a DAG. ENAS' controller is an LSTM RMM with two types of learnable parameters: the controller parameters and the optimization generations parameters. The training of the generated neural networks parameters is done by backpropagation epochs, while the controller parameters are optimized using Adam optimizer [82], for which the gradient is computed using REINFORCE [155]. Liu *et al.* [90] used the DAG concept in Differential Architecture Search (DARTS) to contain the search space for the structure optimization process but with creating a mixture of options at each edge in the graph. NAS, ENAS, and DARTS' DAG is a representation of search space with all the possible structural components (represented in nodes) and all possible operations (represented in edges) and the optimization process generates structures with specific subsets of the search space's structural components and operations.

DARTS relaxes the search space (by applying a softmax over the possible operations between the nodes) to be continuous, and thus, can be optimized with optimized using gradient-descent process. Options were used to probabilistically choose the parameters of the constructed structure. The study reports creating computationally efficient structures in an optimization process that is orders of magnitude faster than NAS. DARTS does not use a controller like NAS and ENAS, making it simpler and applicable to convolutional and recurrent architectures, leading to better performance over NAS and ENAS. Ultimately, the best found structures are achieved, like NAS and ENAS, by jointly learning the architecture (though the optimization of the architectural parameters) and the weights of the generated neural networks. Stochastic Neural Architecture Search (SNAS) [158] replaces NAS' learning through RL with more efficient gradient learning through generic loss. The method devises a search space represented as a matrix with a set of one-hot random variables from a fully factorizable edge distribution, multiplied as a mask to select operations in the DAG. SNAS reports a better performance over NAS and attributes it to the replacement of RL with gradient descent.

While the DAG concept shows good results, the mapping between the network structure and the graph is less direct, which makes the control over the generation process harder. Applying GA and SI can entail indirect encoding but still they are a more transparent

approach compared to the NAS problem.

3.2.2 Neuroevolution Methods

NEAT (NeuroEvolution of Augmenting Topologies) [140] is considered a landmark algorithm in the NE realm. It is a genetic-based algorithm that evolves increasingly complex neural network topologies, while at the same time evolving the connection weights. Genes are tracked using historical markings with innovation numbers to perform crossovers among different structures and enable efficient recombination. Innovation is protected through speciation and the population initially starts small without hidden layers and gradually grows through generations [1, 79, 85]. Experimentations have demonstrated that NEAT presents an efficient way for evolving neural networks for weights and topologies in parallel or separately [140]. Its power resides in its ability to combine all the four main aspects discussed above and expand to complex solutions along the generation process. However NEAT still has some limitations when it comes to evolving neural networks with weights or memory cells for time series prediction tasks, as contended by Desell *et al.* [29].

Desell *et al.* [38] introduced Evolutionary eXploration of Augmenting LSTM Topologies (EXALT), where a simple RNN with LSTM units evolves to the best performing performing structures using a GA where crossovers and mutations are applied to the neural structure represented as a genome. In this evolution process, the neural structure, starting from basic elements, morphs by losing or gaining new structural elements as new offsprings inherit some of their structure characteristics from their ancestors and gain others from the nature-evolution inspired stochastic mutations. EXALT also uses a Lamarckian weight inheritance strategy to enhance the training of the generated structures and speed up the evolution process. Desell also introduced EXACT [26], which evolves convolutional neural network (CNN) structures using a similar methodology. Ororbia *et al.* [109] then introduced Evolutionary eXploration of Augmenting Memory Models (EXAMM). EXAMM expands on EXALT where each of the RNN neurons can either be a Δ-RNN, a GRU, a LSTM, a MGU or a UGRNN unit. It further refines EXALT's mutation operations to reduce hyperparameters using statistical information from parental RNN genomes. Also, where EXALT only used a single steady state population, and EXAMM expands on this to use islands. Islands offer niches for genomes with variant characteristics to thrive and evolve in parallel while allowing for

inter-island crossovers to enhance the genome line of each of those genomic niches to ultimately cover the search space more thoroughly.

Indirect Encoding Neuroevolution

While the constructive neuro-evolution methods based on genetic algorithms have shown impressive successes, the fact that they start from the minimal structure components might put the processes in the position of prematurely converging by falling in local minima in its optimization search process. Due to this, some strategies instead use indirect encoding to design architectures. HyperNEAT (hypercube-based NEAT) [53,139] was one of the first indirect encoding strategies. It generates the weights' pattern as a function of the domain's inputs and outputs geometry. The concept is inspired by how the genotype and the phenotype are mapped in biological genetics. Because a phenotype contains order of magnitude more structural components compared to the genes in a genotype, an indirect encoding exists to map those structural components to the genes that produce it. The idea is that a genotype contains a blue print of repeated patterns that construct structures suited for their roles and functions, making the construction process more abstract and easier as complex structures can be generated from underlying geometric motifs. Therefore, HyperNEAT evolves connective Compositional Pattern Producing Networks (CPPNs) that define connection patterns in ANNs instead of directly evolving the ANN structure. The indirect encoding of HyperNEAT is inspiring for this work as applying ACO entails an abstracted representation for the structural and operational characteristics of the ANN, in which the dynamic actions of the optimization agents are indirectly mapped to the neural structural elements and operations.

Ant Colony Optimization With Continuous Search Domain

Originally ACO was presented as a solution for a discrete search space problem (the TSP), but eventually ACO was adapted to continuous search domains. In [134], it was shown that ACO "can be adapted to continuous optimization without any major conceptual change to its structure." Their method, which they called $ACO_{\mathbb{R}}$, essentially showed how to shift the discrete search of the ACO's agents to a continuous one by applying probability density function (PDF) to iteratively construct solutions for a

given problem.

ACO Variable was proposed by Kuhn [83] as a continuous domain ACO algorithm in which ants use a Gaussian distribution to sample solutions from randomly selected set of candidate steps and from those steps, the ones with higher expected pheromone are chosen with greater probability. Xiao *et al.* [157] introduced a hybrid ant colony optimization for continuous domains (HACO), which uses continuous-domain ACO with a differential evolution. The algorithm follows the same methodology introduced by Socha and Dorigo [134] but applies differential evolution to increase the search diversity to alleviate being trapped in local minimas. Gupta *et al.* [61] used $ACO_{\mathbb{R}}$ to optimize transistor size optimization in digital circuits following the algorithms described by Socha and Dorigo [134].

Those studies build a number of solutions by initially assigning values randomly to the variables of the problem then placing those N solutions in a sorted array (*i.e.*, a population). Pheromone values are assigned to those solutions based on their rank in the population. The pheromone values are used to select one of the solutions based on a probability distribution that depends on the pheromone values. After that, a new solution is obtained by using a Gaussian distribution bounded by the selected solution variables values. Finally the population is updated with the newly generated solution and the worst solution in the population is removed to keep its size constant.

Another work by Bilchev *et al.* [8] introduced Continuous Ant Colony Optimization (CACO) where agents emerge from a nest in vectors-like patterns. Through iteration, agents evolve their vectors' direction based on the trail of pheromones left by successful agents' movement in previous iterations. The study states that "an obvious limitation of the current algorithm is the lack of a model for the exhausting of the food source. Thus, it is possible for search agents to repeat already tracked and assumed exhaust paths."

While the aforementioned studies search a continuous search space that does not represent a graph, this research addresses continuous ACO for graph optimization problems with undefined bounds.

RNN Regularization vs. ACO Optimization

Srivastava *et al.*have demonstrated the ability to utilize Dropout, a popular method for regularization of convolutional neural networks [136], to be applied as a regularizer for RNNs [164]. This work has shown strong results in reducing overfitting when utilizing large RNNs. As dropout randomly drops out connections during the forward pass of the backpropagation algorithm, it effectively trains the network over randomly sampled subnetworks of the fully connected architecture. As each forward path is a different randomly selected network, this forces the trained weights to become more robust and serves to reduce overfitting.

While the dropout approach is highly successful for classification problems, such as those presented in Zaremba *et al.*'s work [164], which can easily be overfit – this work focuses on time series data prediction. In all tested architectures in previously published work [39], the RNNs have not come close to overfitting on the training data, but rather the problem has been the effective training of the network given the highly challenging prediction task. The MC-ACO approach described in this previous work and as the first ACO algorithm for NAS developed in this thesis focuses on finding the best subset of connections to use in an LSTM RNN, which makes training them more efficient and effective – using a fixed sub-topology for an entire training process, as opposed to randomly dropping out connections in each forward pass, as done by dropout. In addition, the input parameters used in this work are highly non-seasonal and acyclic, which makes the RNN overfitting hard to occur (see Section 5.1).

Chapter 4

Methodology

This study is concerned with investigating various nature-inspired neural architecture search (NI-NAS) algorithms based on ant colony optimization (ACO) and their ability to perform RNN topology optimization for real-world applications based on time series data prediction. The parameters in the operational time-series data extracted from industrial applications are correlated and dependent on each other, but the data are also highly non-linear, non-stationary, and non-seasonal. This is due to the fact that these systems and their sub-systems are engineered systems rather than being natural systems, where the high-nonlinearity, non-stationary, and non-seasonality are more likely to be present in an engineered system, and where the parameters in natural systems tend to be more harmonic. To make inferences and predictions using these data sets, this work utilizes deep learning due to the limitations of statistical methods described previously in Chapter 2.

While deep RNNs have the strongest potential for developing accurate models for these tasks, as mentioned in Chapter 1, a massively connected network can be a burden to the training process instead of enhancing it [39], while at the same time being computationally wasteful. The depth of the used models entails structural elements may produce and amplify noise rather than help the model to learn about the problem. Additionally, the RNN inputs might themselves be a source of noise when the data is corrupted or some parameters are not actually contributing to the solution. Therefore, a structural optimization effort is required to potentially eliminate those structural elements based on optimization criteria. Figure 4.1 illustrates visualization for a fully connected RNN

with limited input, hidden, and output nodes. The schematic has recurrent connections spanning limited time-lags, which feed data from the nodes at previous time steps to the current time step. To address this challenge, the study investigates three algorithms based on ant colony optimization (ACO) [31] to optimize and design recurrent neural networks.

A goal of these strategies is both finding an RNN structure requiring minimal computational resources while still producing highly accurate results. The process also offers an automated, empirical strategy for examining input parameters to see which of them are actually contributing to the application problem. This inference can be done by looking into the input nodes in the optimized structure and observe how they are connected to the rest of the network. As the number of connection fanning out from a node is mostly in direct proportion to the amount of information flowing out of it – except when the weight of connection is significant small – fewer connections can be an indication that the parameter which this input node represents is not effective for the problem under investigation.

In nature, the pheromone is one primary driver of how ants communicate with each other, the traces of which allow the collective to "know" of potential food sources ensuring the survival of the colony in the long term. When an ant finds food, the ant will start marking the path it takes to return back to the colony, the pheromone trace of which other ants will then subsequently follow. The work in this thesis draws from this chemical-driven behavior in order to design algorithms that simulate traces on artificial neural network (ANN) topologies. Specifically, these traces will be simulated by introducing an additional, dynamic scalar weight (or importance value) assigned to a given (neural) synapse, which will bias any given ant agent to favor selecting some possible (more rewarding) synaptic pathways in the ANN over others.

The mathematical representation of this process of edge selection, related to pheromone values deposited during ACO can be described as:

$$edges_{neural\_network} = fn(\tau_c \mid c \in superstructure)$$

$$\tau_c = fn(fitness)$$

where: τ_c is the pheromone value on a connection of the super structure, and c is a connection in the super structure. In essence, edges are sampled by a function that depends on the pheromone levels in the ACO search space, and the pheromone values

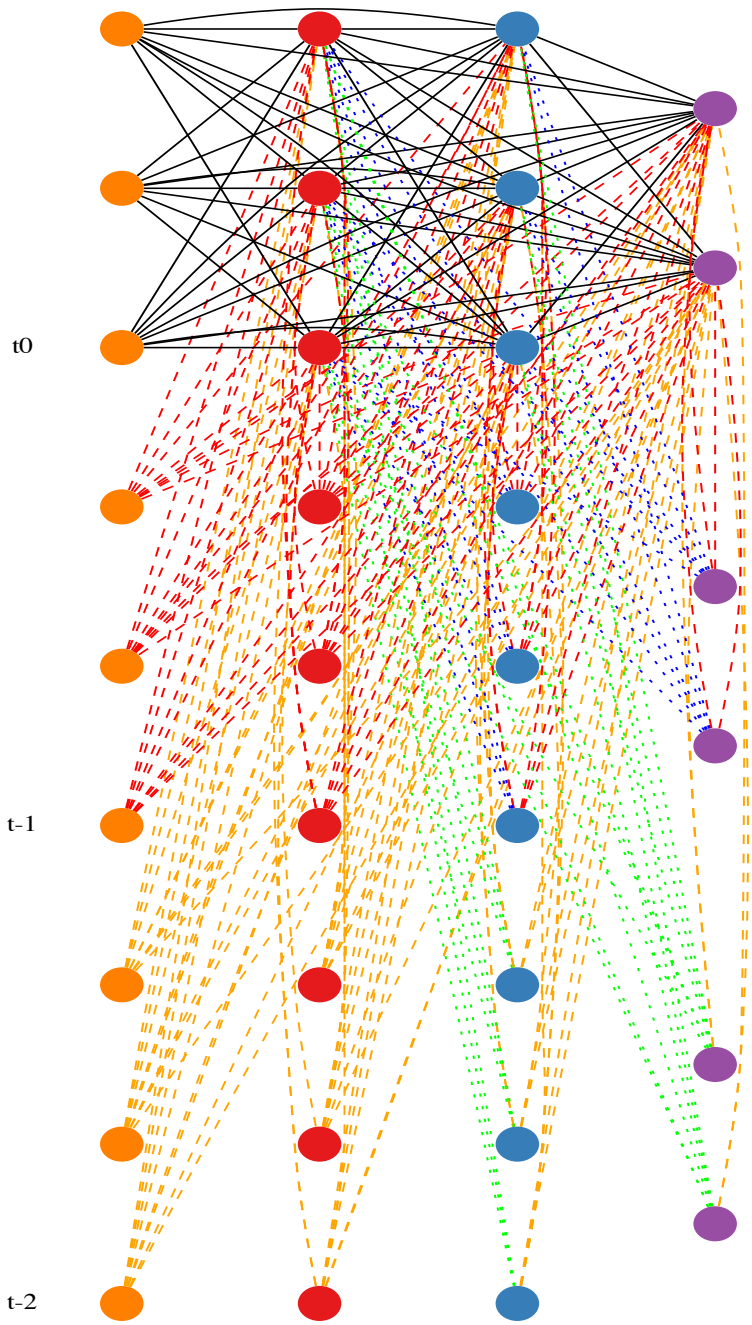


Figure 4.1: *A schematic for a completely connected recurrent neural network. The structure consists only of 3 input nodes, 2 hidden layers with 3 nodes in each, and 2 output node. The recurrent connections come from only 2 time lags. The input nodes are orange circles, hidden layers nodes are red and teal circles, and the output nodes are violet circles. Edges are solid black lines. Forward recurrent edges are dashed lines: red from* $t-1$ *and orange from* $t-2$*. Backward recurrent Edges are dotted lines: blue from* $t-1$ *and green from* $t-2$*.*

on each connection in the ACO search space are function in the fitnesses of previously evaluated networks.

Using ACO as basis for NAS in this work was done in three stages. First, an LSTM RNN substructure was optimized and populated over all the LSTM cells in the architecture. Second, a more holistic approach was used by applying the optimization process to the entire structure. A set of heuristics were used to perform the optimization in a discrete search space that is represented with a massively connected superstructure. Third, the discrete search space was replaced by a continuous one in an easier to use method that had less hyperparmaters.

4.1 Using ACO to Optimize LSTM Cells

The first exploration of ACO for NI-NAS in this work started by optimizing a particular sub-structure of an LSTM RNN and re-using the optimized connection sub-structure over all the gates of an LSTM cell for all the network's time steps. This previous work was, memory cell ant colony optimization (MC-ACO) was published in two works [39,42] improving on work investigating the use of LSTM RNNs to predict aircraft engine vibration [41].

To evaluate this strategy, three LSTM RNN architectures were designed to predict engine vibration 5 seconds, 10 seconds, and 20 seconds in the future. Each of the 15 selected aviation Flight Data Recorder (FDR) parameters were represented by a node in the inputs of the neural network and an additional node was used for a bias. Each neural network in the three designs consists of modified LSTM cells that receive both an initial input of flight data at some time in the past or the output from a cell in the lower layer, and the output of the previous cell in the same layer, as inputs (see Figure 4.2). Each cell has three gates to control the flow of information through the cell and accordingly, the output of the cell. Each cell also has a cell-memory which is the core of the LSTM RNN design. The cell-memory allows the flow of information from the previous states into the current predictions.

The gates that control the flow are shown in Figure 4.3. They are: *i)* the *input gate*, which controls how much information will flow from the inputs of the cell, *ii)* the *forget gate*, which controls how much information will flow from the cell-memory, and *iii)* the

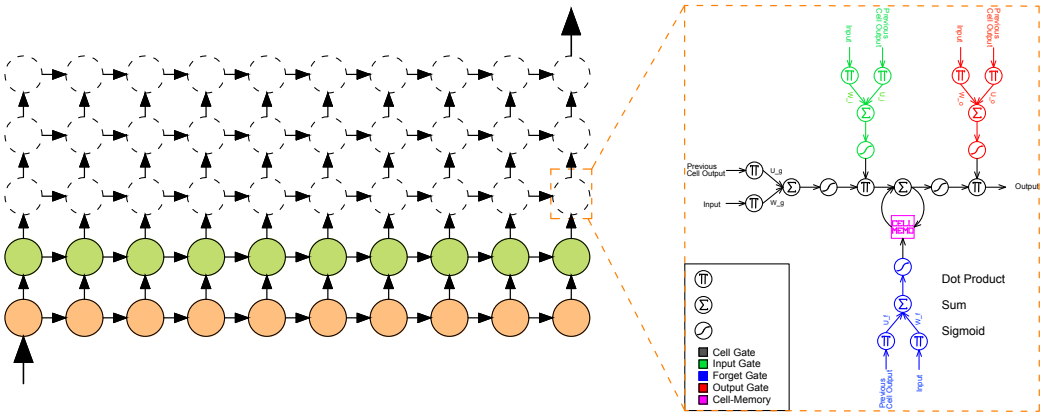


Figure 4.2: A general overview of the design of an LSTM-RNN.

output gate, which controls how much information will flow out of the cell. This design allows the network to learn not only about the target values, but also about how to tune its controls to reach the target values.

All the utilized architectures follow the common LSTM RNN designs shown in Figure 4.2 and 4.3. However, there are two variations of this common design used in the utilized architectures, shown in Figures 4.4 and 4.5, with the difference being the number of inputs from the previous cell. Cells that take an initial number of inputs and output the same number of outputs are denoted by *'M1'* cells. As input nodes are needed to be reduced through the neural network, the design of the cells are different. Cells which perform a reduction on the inputs are denoted by *'M2'* cells.

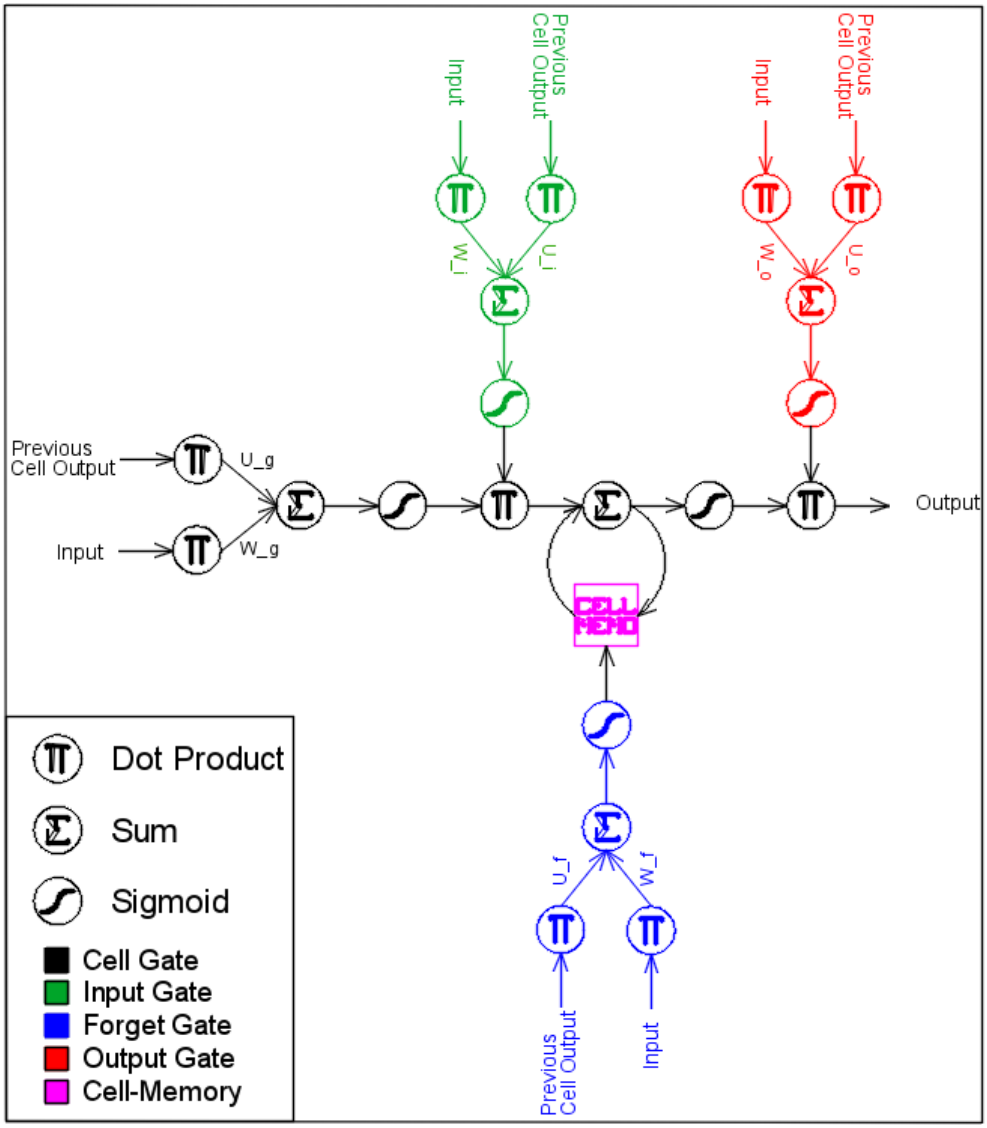


Figure 4.3: *LSTM cell design*

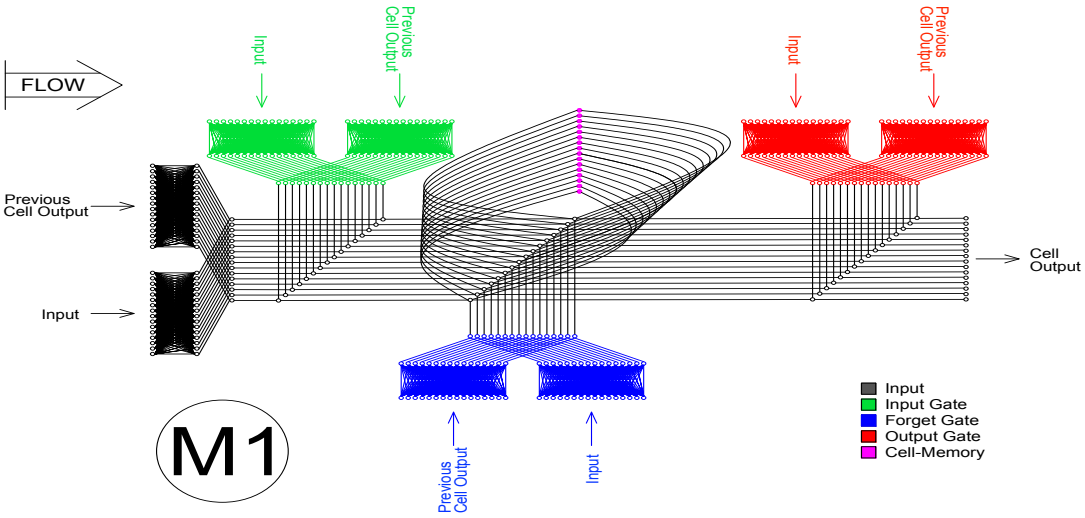


Figure 4.4: *Level 1 LSTM cell design*

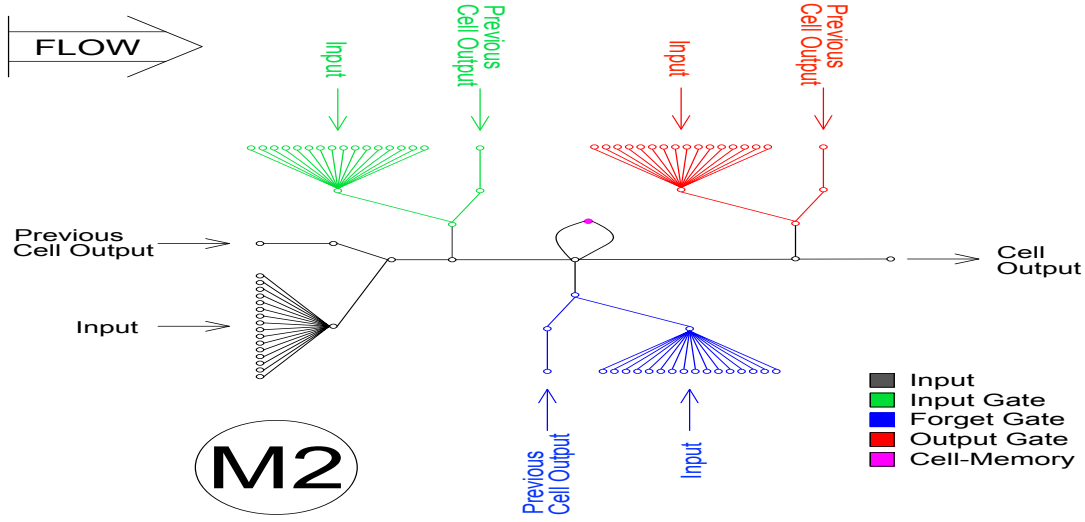


Figure 4.5: *Level 2 LSTM cell design*

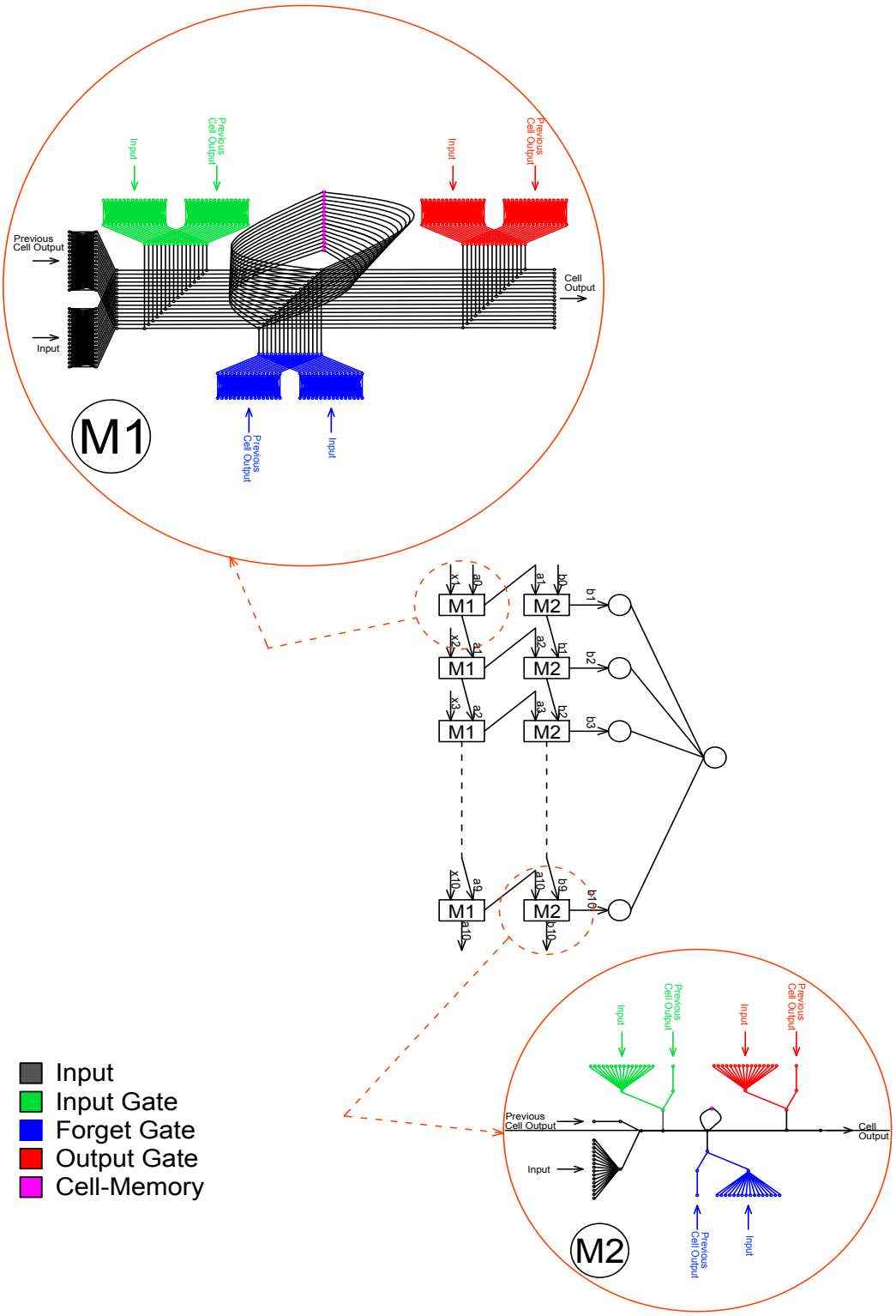


Figure 4.6: *How the internal structures of the modified LSTM cells are used in the overall architecture.*

LSTM RNN Forward Propagation Equations

The equations used in the forward propagation through the neural network are:

$$i_t = Sigmoid(w_i \bullet x_t + u_i \bullet a_{t-1} + bais_i) \tag{4.1}$$
$$f_t = Sigmoid(w_f \bullet x_t + u_f \bullet a_{t-1} + bais_f) \tag{4.2}$$
$$o_t = Sigmoid(w_o \bullet x_t + u_o\bullet \tag{4.3}$$
$$g_t = Sigmoid(w_g \bullet x_t + u_g \bullet a_{t-1} + bais_g) \tag{4.4}$$
$$c_t = f_t \bullet c_{t-1} + i_t \bullet g_t \tag{4.5}$$
$$a_t = o_t \bullet Sigmoid(c_t) \tag{4.6}$$

where (see Figure 4.3):

i_t: input-gate output

f_t: forget-gate output

o_t: output-gate output

g_t: input's sigmoid

c_t: cell-memory output

w_i: weights associated with input and input-gate

u_i: weights associated with previous output and input-gate

w_f: weights associated with input and forget-gate

u_f: weights associated with previous output and forget-gate

w_o: weights associated with input and output-gate

u_o: weights associated with previous output and the output-gate

w_g: weights associated with the cell input

u_g: weights associated with previous output and the cell input

and the formula of the sigmoid function is:

$$Sigmoid(\alpha) = \frac{1}{1 + e^{-\alpha}} \tag{4.7}$$

4.1.1 LSTM RNN Architectures

The three architectures are as follows, with the dimensions of the weights of these architectures shown in Table 4.1 and the total number of weights shown in Table 4.2.

Architecture I As shown in Figure 4.7a, the first level of the architecture takes inputs from ten time series (the current time instant and the past nine). It then feeds the second level of the neural network with the output of the first level. The output of the first level of the neural network is considered the first hidden layer. The second level of the neural network then reduces the number of nodes fed to it from 16 nodes (15 input nodes + bias) per cell to only one node per cell. The output of the second level of the neural network is considered the second hidden layer. Finally, the output of the second level of the neural network would be only 10 nodes, a node from each cell. These nodes are fed to a final neuron in the third level to compute the output of the whole network.

The dimensions of the weights matrices and vectors of this architecture are shown in Table 4.1. The total number of weights are is shown in Table 4.2. Figures 4.8 and 4.9 provide an overview of architecture I, as it has a large number of connections (21,170). Figure 4.8 shows the overall design of how the LSTM cells are connected, and then Figure 4.9 displays all the connections within a single time step of the full LSTM RNN. As a whole, there are 10 different instances of Figure 4.8, each connected as specified in Figure 4.9.

Architecture II As shown in Figure 4.7b, this architecture is almost the same as the previous one except that it does not have the third level. Instead, the output of the second level is averaged to compute the output of the whole network. The dimensions of the weights matrices and vectors of this architecture are shown in Table 4.1. The total number of weights are shown in Table 4.2.

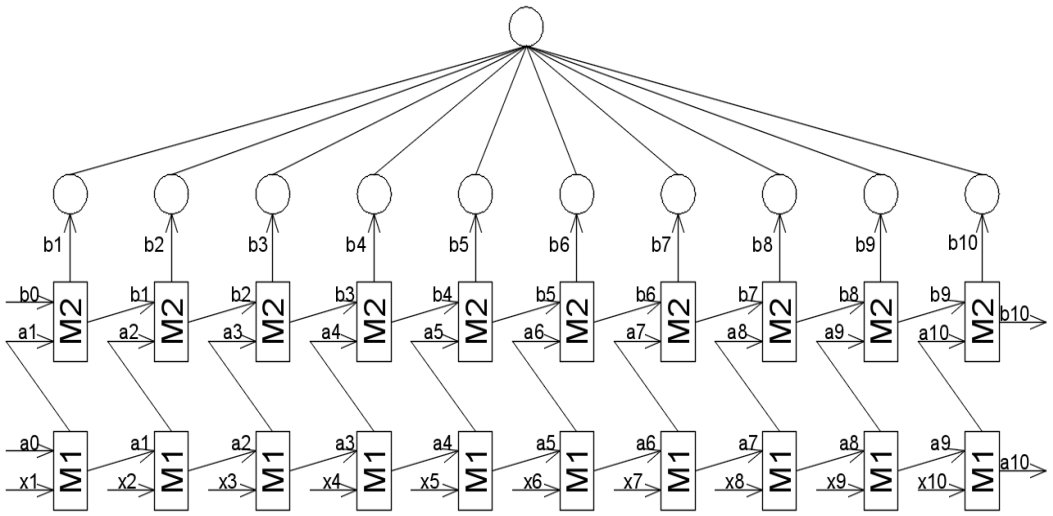
Architecture III Figure 4.7c presents a deeper neural network architecture. In this design, the neural network takes inputs from twenty time series (the current time instant and the past nineteen) as the first level. It feeds the second level of the neural network with the output from the first level. The second level does the same procedure as the first level giving a chance for more abstract decision making. The output of the second level of the neural network is considered the first hidden layer and the output of the second level is considered the second hidden layer. The third level of the neural network then reduces the number of nodes fed to it from 16 nodes (15 input nodes + bias) per cell to only one node per cell. The output of the third level of the neural network is considered the third hidden layer. Finally, the output of the third level of the neural network is twenty nodes, a node from each cell. These nodes are fed to a final neuron in the fourth level to compute the output of the whole network. The Dimensions of the weights matrices and vectors of this architecture are shown in Table 4.1. The total number of weights are shown in Table 4.2.

4.1.2 Evolving LSTM RNN Cells using Ant Colony Optimization

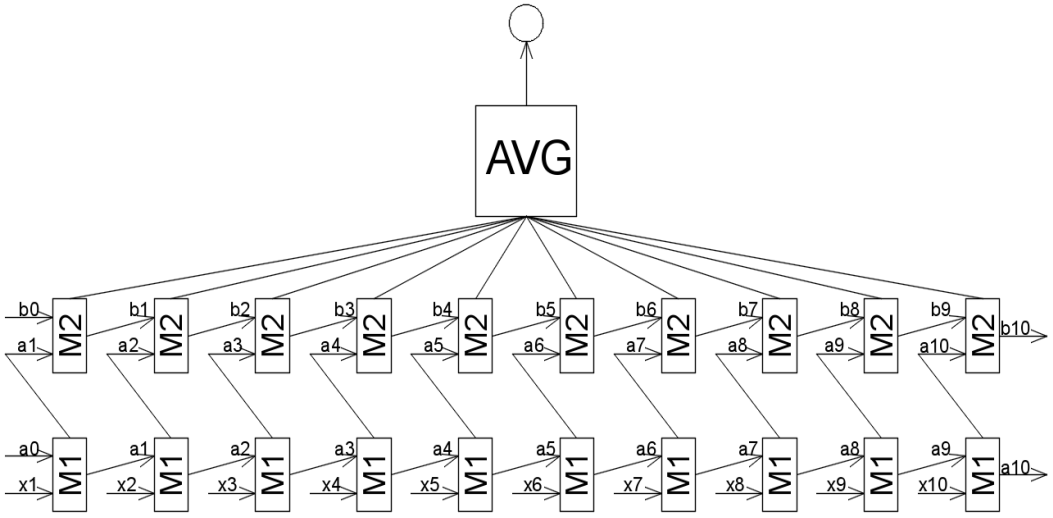
Although the results from architecture I were promising, there was still room for further optimization in that the network may have excessive connections which confound accurate predictions and that the structure could be further optimized. A particular concern was that some connections could cause noise in the obtained results and ultimately would drift the results from their most optimum values, as this had been shown in the initial one layer feed forward neural networks with certain input parameters. The goal of using the ant colony optimization strategy is to evolve the structure of the LSTM cells, encouraging more diverse networks and selecting the topologies that give the best performance.

The ACO algorithm operates on the fully connected inputs to the M1 and M2 cells, as shown in Figures 4.4 and 4.5. Each M1 cell has eight 16x16 input gates, four of which take the input from the previous cell in the same layer, and four of which take the input from the time series or the cell in the lower layer. Each M2 cell has eight 8x1 input gates, four of which receive input from the previous cell in the same layer and four from the cell in the lower layer.

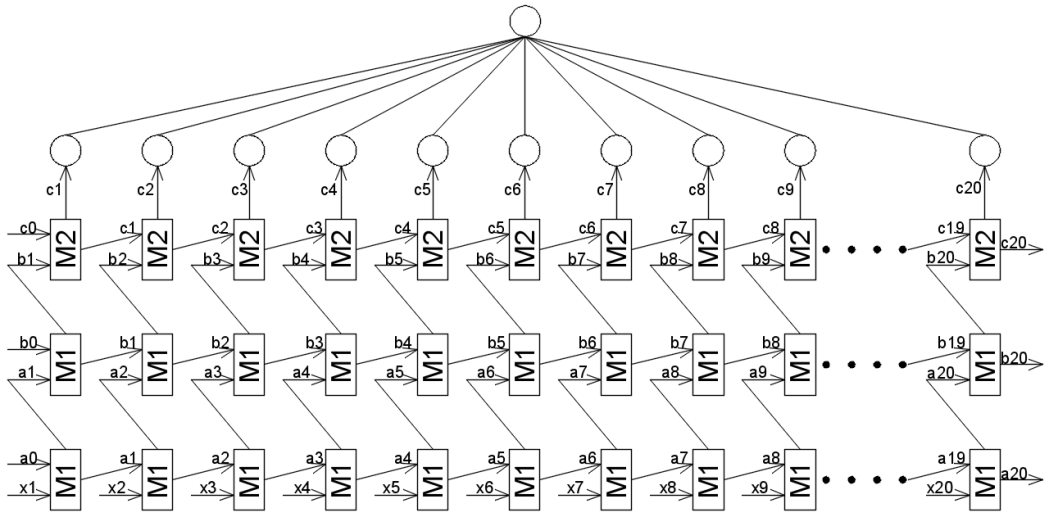
The algorithm begins with a fully connected gate that will be used by the ants each time



(a) *Architecture I*



(b) *Architecture II*



(c) *Architecture III*

Figure 4.7: *The LSTM-RNNs Architectures explored in this thesis.*

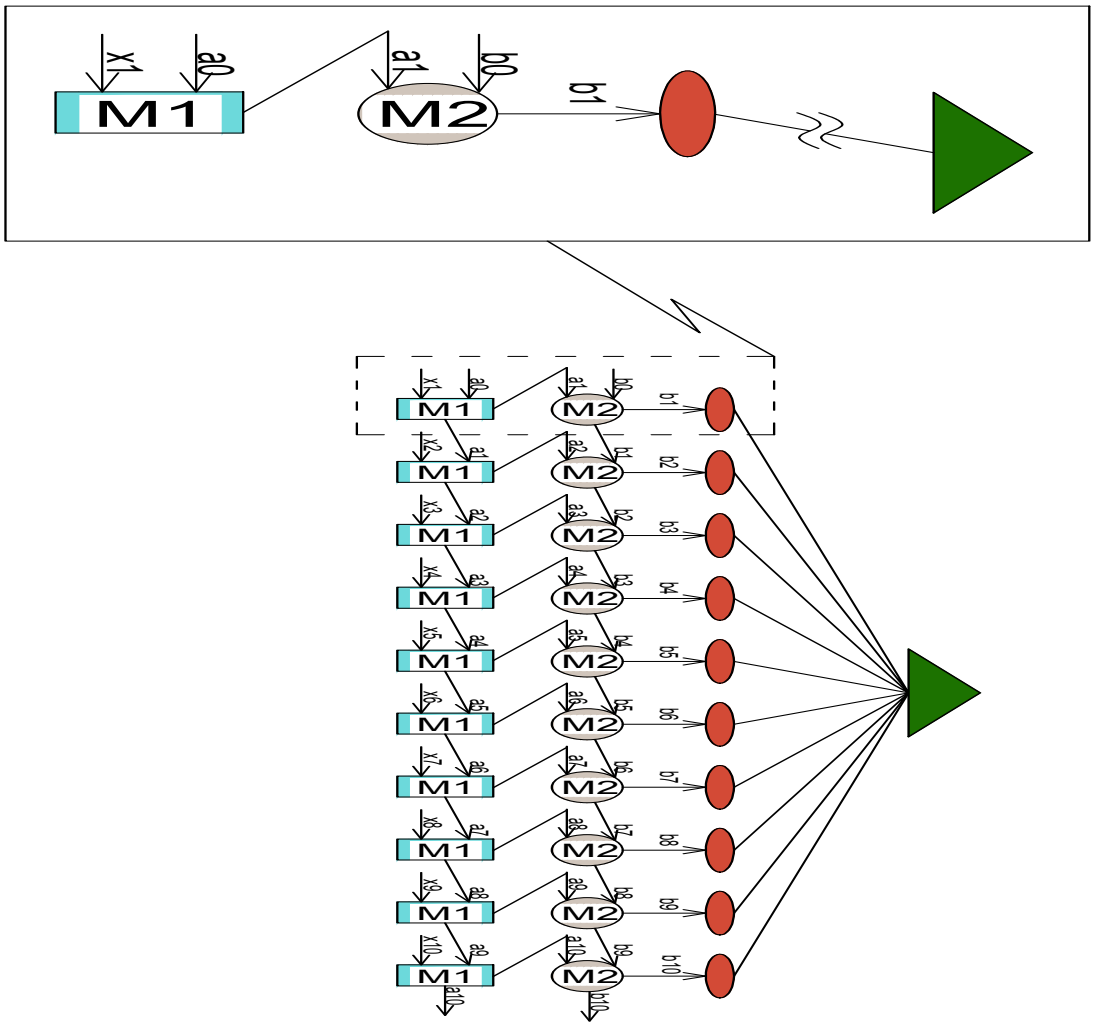


Figure 4.8: *One time-step in the Architecture Full Structure*

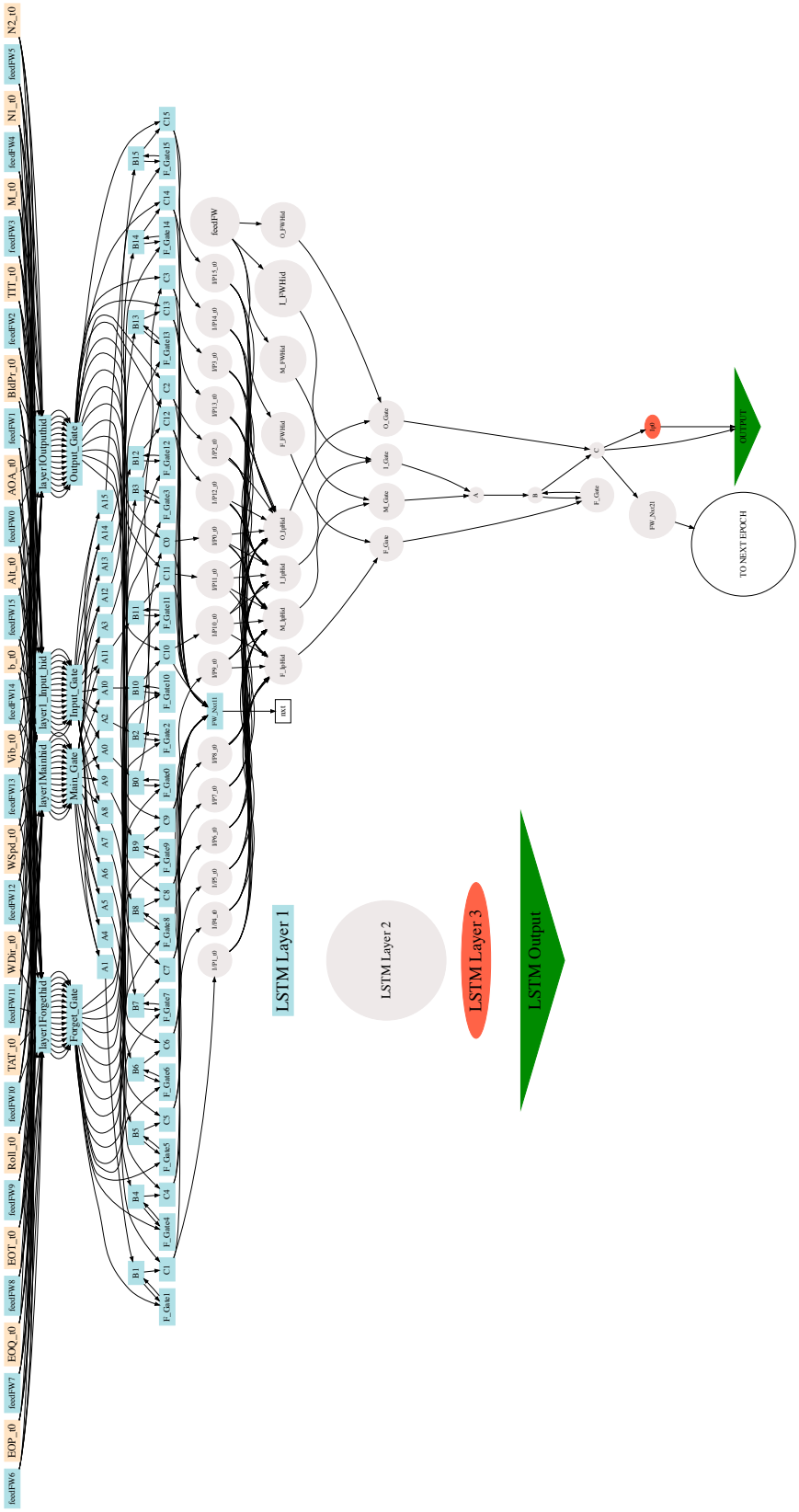


Figure 4.9: *One time-step in the Architecture Full Structure*

Table 4.1: Architectures Weights-Matrices Dimensions

Architecture I

	w_i	u_i	w_f	u_f	w_o	u_o	w_g	u_g
Level 1	16×16	16×16	16×16	16×16	16×16	16×16	16×16	16×16
Level 2	16×1	1×1	16×1	1×1	16×1	1×1	16×1	1×1
Level 3	16×1							

Architecture II

	w_i	u_i	w_f	u_f	w_o	u_o	w_g	u_g
Level 1	16×16	16×16	16×16	16×16	16×16	16×16	16×16	16×16
Level 2	16×1	1×1	16×1	1×1	16×1	1×1	16×1	1×1

Architecture III

	w_i	u_i	w_f	u_f	w_o	u_o	w_g	u_g
Level 1	16×16	16×16	16×16	16×16	16×16	16×16	16×16	16×16
Level 2	16×16	16×16	16×16	16×16	16×16	16×16	16×16	16×16
Level 3	16×1	1×1	16×1	1×1	16×1	1×1	16×1	1×1
Level 4	16×1							

Table 4.2: Architectures Weights Matrices' Total Elements

Architecture I	Architecture II	Architecture III
21,170	21,160	83,290

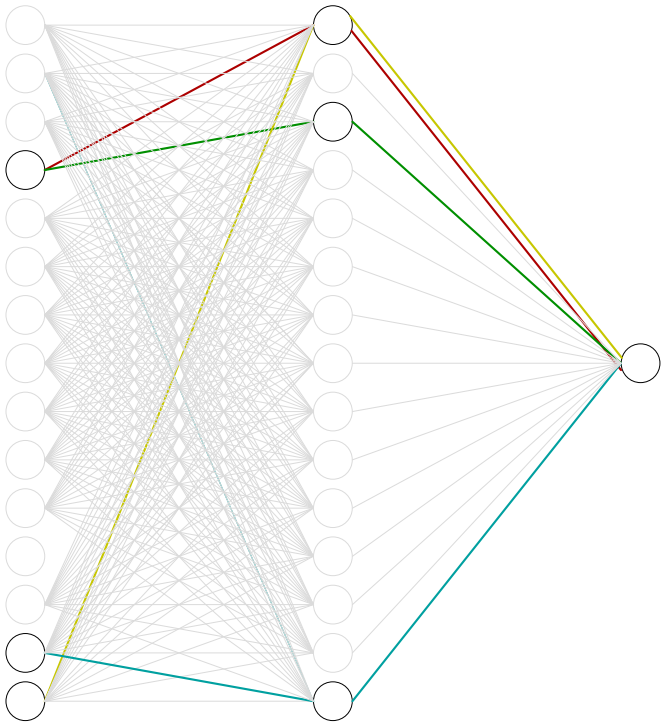


Figure 4.10: *A schematic of an artificial neural network structure found by AOC.*

to generate new paths for new network designs. Paths are selected by the ants based on pheromones – each connection in the network has a pheromone value that determine its probability to be chosen as a path. Given a number of ants, each one will select one path from the fully connected network. All the paths selected from all the ants are then collected, duplicated paths are removed and a design network is generated based on the new cell topology. Figure 4.10 shows an example on an M1 cell, assuming four ants choosing their paths on an input gate to an M1 cell, which generates a subgraph from the potentially fully connected input gate. The same ACO generated topology is used for each of these 8 input gates. Figure 5.8a provides an example of the best found ACO optimized M1 cell.

In detail, the paths generated by ACO are used in the connections between the *"Input"* and the hidden layer neurons that follow it, and the *"Previous Cell Output"* and the hidden layer neurons that follow it. The connections between the *"Input"* and the hidden layer neurons that follow it are shown in the first level cells' (Figure 4.4 "M1") in BLACK color, GREEN color, BLUE color, and RED color at the gates of the cell.

Once a hidden node in the first level cell is reached by an ant, the connection between this node and the output node shown in the second level cells' Figure 4.5 "M2" in BLACK color, GREEN color, BLUE color, and RED color, will automatically be part of the evolved mesh because the ant will not have any other option to reach the output node except through that single connection.

The same generated mesh is used at all the gates: Main Gate, Input Gate, Forget Gate, and Output Gate at the "M1"cells and "M2" cells at all the time-steps in the LSTM RNN Architecture I as shown in Figure 4.7a. In other words, regardless of the LSTM RNN time-step, whenever there is a transition without data reduction: the first set of connections in the generated mesh is used, and whenever there is a transition with data reduction: the second set of connections in the generated mesh is used.

Section 5.2 presents results comparing the MC-ACO optimized topologies to traditional hybrid methods (NOE, NARX, NBJ models) as well as to the standard dropout normalization method.

4.1.3 Expanding NOE, NARX, and NBJ for Multiple Time Step Input

The model sdiscussed in Chapter 2.3 were used to compare their results to the results obtained from using ACO. These results are presented in Chapter 5.2. Following are how each model was prepared for the problem:

Expanded NOE Model

The actual vibration values are fed as an input along with the current instance parameters and lag inputs. To make the model more comparable to the architectures used in this study, the parameters fed are the same used in the proposed LSTM RNN architectures to predict the vibration value in 10 seconds in the future, *i.e.*, they utilize the previous 10 seconds of input data, instead of just the current input data. The NOE does not have actual recurrent inputs, as it instead includes the actual prediction value as input instead. The vibration has been included as an input parameter in all models utilized, so the NOE model is no different than a traditional feed forward network.

Expanded NARX Model

This network, has been updated in a similar way to the NOE network. The previous 10 seconds of input data are utilized, and the previous 10 output values are fed to the network as recurrent inputs. Traditionally in the NARX model, the weights for recurrent connects are fixed constants [106], and therefore their corresponding inputs are not considered in the gradient calculations and these weights are not updated in the training epochs. The NARX network depicted in Figure 2.10 was used. However, the output of the cost functions in the training iterations of this implementation froze at a constant value, indicative of a case of vanishing gradients. Accordingly, the study allowed for the recurrent weights to be considered in the gradient calculations in order to update the weights with respect to the cost function output.

Expanded NBJ Model

As previously noted, this network is not feasible for prediction past one time step in the future in an online manner, as it requires the actual prediction value and error between it and the predicted value to be fed back into the network. However, as this work dealt with offline data, the actual future vibration values, error, and the output were all fed to the network along with the current instance parameters and lag inputs. As in the other networks, the values for the previous 10 time steps were also utilized.

4.2 Using ACO to Optimize Neural Structure in a Discrete Domain

After successfully applying ACO to optimized the internal structure of an memory-based cell (which is a substructure), the next natural step was to investigate the application of the concept to a complete RNN structure. ANTS was developed Ant-based Neural Architecture Search to address this problem. At a high level, in ANTS, the individual ant agents operate on a single massively connected "superstructure" (a small one is shown in Figure 4.1), which contains all possible ways that neural network nodes may connect with each other both in terms of structure, *i.e.*, all possible feed forward pathways that start from the input nodes and end at the output nodes, and time, *i.e.*, all

possible recurrent connections that span many multiple time delays[1]. As done in ACO, ants choose to move over connections between nodes, probabilistically as a function of a simulated chemical known as the "pheromone", which is placed on connections by ants based on how well they have been utilized to generate candidate RNNs.

ANTS was developed as an asynchronous parallel system for use on high performance computing resources, which has a *main* process that maintains the colony information and *worker* processes to (locally) train the RNNs. Algorithm 1 depicts a high-level pseudocode for the ANTS. This parallel implementation is asynchronous, the *main* process generates new RNNs as needed for *worker* processes (which operate on separate, dedicated CPU or GPU resources) and updates colony information and pheromones as trained RNN results are returned. This asynchronous design is known as work-stealing parallel programming and is depicted in Figure 4.11. The result is a naturally load balanced algorithm with high scalability. From the overall superstructure, which the ant agents exclusively operate on, RNN subnetworks are extracted (as dictated by the current pheromone trace network available at the current simulation time step, which yields a map of nodes and connecting synapses, both recurrent and feed forward, visited by the ant agents) and sent to *worker* processes. The *worker* processes train the extracted RNNs locally with only a few epochs of back-propagation through time (BPTT). After a particular *worker* is done locally training a RNN subnetwork, the candidate's weight values and cost (fitness) function (measure on a validation subset of data) are communicated back to the swarm and superstructure (housed in the *main* process), adjusting the pheromone trace network and affecting future ant agent traversal behavior.

Within the main process itself, ANTS operates by having a fixed number of ant agents traverse the neural superstructure. Ants choose to move over connections between nodes randomly, but they are probabilistically biased towards connections with higher simulated "pheromone" values. Pheromone deposit values are periodically evaporated to prevent the search process from becoming stuck in local minima. Interestingly enough, the modification of the evaporation function can be considered a way in which one could encode certain priors into the ANN itself.

[1]Note that this superstructure is more connected than a standard fully connected neural network – each layer is also fully connected to each other layer as well, allowing for forward and backward layer skipping connections, with additional recurrent connections between node pairs for each time skip allowed.

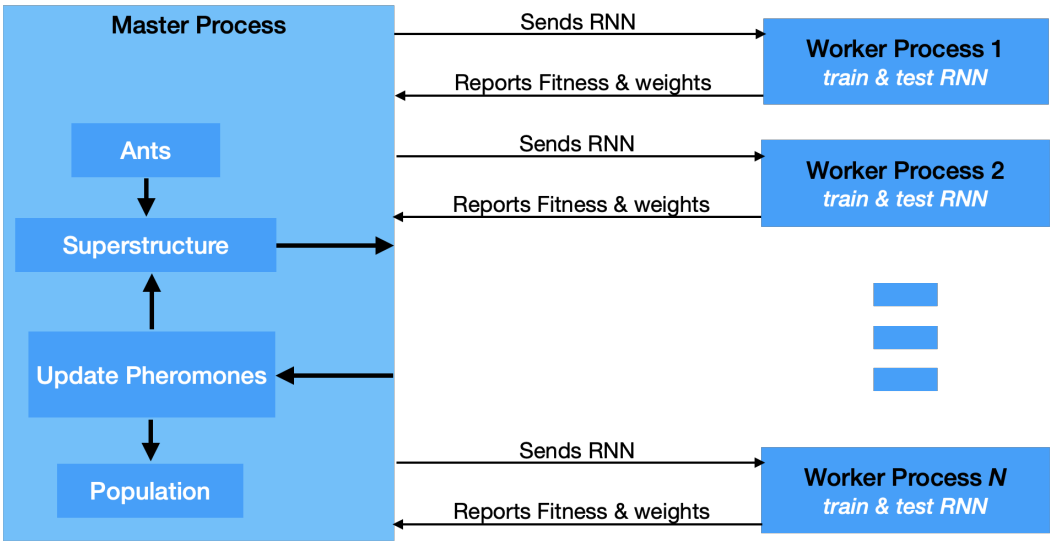


Figure 4.11: *In the work-stealing asynchronous parallel computing design, a main process takes care of generating RNN's structures, updates the evolution population, and rewards the ants by pheromone depositions when they build a good performing structure, which are trained by workers.*

Within the framework of ANTS, variations of its various underlying mechanisms were investigated. These include the use of communal intelligence by sharing the best weights found among the colony, allowing ant agents to also select from multiple memory cell types as opposed to operating exclusively with simple neurons, introducing specialized ants that have different graph traversal strategies, and constraining ant movement and manipulating the pheromone evaporation function in order to encourage the discovery of sparse RNN topologies. One particularly crucial element in the ANTS procedure is the introduction of different ant agent types or species, which is inspired by how real ants specialize to act according to specific roles to serve the needs of the colony [113]. Specifically, different ant agent's designs were considered to serve specific roles in constructing parts of candidate RNN subnetworks – some ants exclusively traverse feed forward synaptic pathways while others only explore recurrent synaptic pathways.

The ANTS implementation contained a number of sub-heuristics which were investigated to improve performance, such as communal weight sharing, memory cell selection, altering graph traversal with different ant species, pheromone update and evaporation strategies, which are described in the following sections.

4.2.1 Communal Weight Sharing

Edges and recurrent edges' weights can be randomly initialized each time a new RNN is generated by the ants. However, initializing parameters this way requires local tuning (via BPTT) for many epochs for the RNN to reach suitable generalization error, as they do not make use of any information gained by prior trained RNN candidates. Further, the reuse of prior trained weights (*i.e.*, epigenetic or Lamarckian weight initialization) can significantly speed up the neuro-evolution process and result in better performing, smaller ANNs in general [27].

To apply similar prior knowledge in ANTS, the algorithm turns to utilizing a "communal weight sharing" strategy. Each edge in the ant swarm's connectivity super-structure also tracks a weight value in addition to its pheromone value. These weights are randomly initialized uniformly $U(-0.5, 0.5)$. Each time a generated RNN performs well, the weights of its best performance, as measured on a validation data subset, are used to update the shared weight values in the swarm's super-structure.

Formally, Φ is defined as a function of the population's best and worst evaluated RNN fitness, where W_{colony_i} is the colony's edge weight, W_{RNN_i} is the corresponding neural network's edge weight, fit_{pop_best} is the population's best fitness, and fit_{pop_worst} is the population's worst fitness. Weight initialization then proceeds as follows:

$$x = \frac{fit_{new} - fit_{pop\_best}}{fit_{pop\_worst} - fit_{pop\_best}} \tag{4.8a}$$

$$\Phi(fit_{new}) = min\Bigg(max\Big((1-x), 0\Big), 1\Bigg) \tag{4.8b}$$

$$W_{colony_i} = \Phi W_{RNN_i} + (1 - \Phi) W_{colony_i}. \tag{4.8c}$$

With respect to the function Φ, two variations were investigated. The first variant, as shown in Equation 4.8, used the fitness of the RNN used to update the weights to determine how much these new (locally found) weight values effect those of the colony. The second variant of Φ was set to a predetermined constant instead of being calculated or adjusted by fitness. This process essentially allows for a running average (either with a fixed update or dynamic update based on fitness) of the best weights found for each connection in the superstructure. When a new RNN is generated, it uses the current weight values in whatever edges that were extracted from the superstructure on the *main* process. The process allows for the colony to share information about the best

weights found for each connection, adapting them in a manner similar to a running average as new best candidate RNNs and weights are found.

4.2.2 Memory Cell Selection

For any particular node in the super-structure, ANTS also has the ability to utilize the pheromones present to select which memory cell type a particular node will be in the generated network. A node could be chosen to be either an LSTM [65], a GRU [20], an MGU [165], a UGRNN [22], or a Δ-RNN cell [112]. The formulations of these memory cells are discussed in Chapter 2.2.3. Pheromones are deposited and updated for each of these memory cell possibilities as described below.

4.2.3 Altering Graph Traversal with Ant Species

As mentioned above, various strategies were explored to guide ant traversal over the connectivity superstructure. Inspired by role specialization in real colonies, ant agents were implemented to explore the connectivity graph in specific ways. First, a generic ant agent, called the *standard ant*, was allowed to traverse through the massively connected colony superstructure in an unbiased manner. This generic agent, in essence, recovers the standard simple ant agent in classic ACO, which has complete freedom to explore any piece of a given graph structure.

However, it became quickly apparent that this type of ant would get "stuck" in the network, generating a significantly high number of recurrent connections before finally reaching an output node (explained in Figure 4.12). That is the RNN candidates extracted for local fine-tuning were rather dense, and in turn, compute-heavy (featuring many extraneous parameters as a characteristic of over-parameterized models).

Why do standard/simple ants get stuck or meander too long in the superstructure? In the superstructure, nodes (especially at the final hidden layer) have the option of selecting potential backward recurrent paths, which significantly outnumber the number of potential forward moving paths (see Figure 4.12). Assuming that each connection has an equal number of pheromones (which is a standard setting for pheromone initialization), agents will circle around the colony using these backward paths, yielding RNN candidates with very dense recurrent structure.

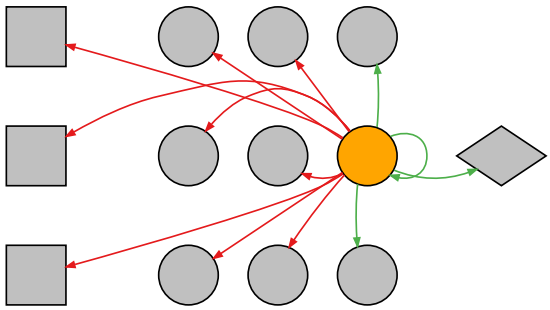


Figure 4.12: *Potential paths that an ant can take from a given node (in orange) with the massively-connected superstructure. The number of recurrent paths (red) far outnumber the forward paths (green). This problem is exacerbated as the possible recurrent time scale increases, which results in multiple backward recurrent connections for each red connection, each going back a different number of time steps in the past.*

To prevent this problem, the first tactic was to alter the pheromone deposit function by adding extra pheromones to forward paths upon initialization as well as after every pheromone update. If the total number of pheromones on the forward edges out of a node was less than 75% of the total number of pheromones on the recurrent edges out of the node, the pheromones on each forward node were multiplied by the ratio of the total sum of outgoing recurrent edge pheromones over the total sum of the outgoing forward edge pheromones. This biasing method yielded better proportions of forward and backward paths.

Even with this forward path bias added to the pheromone deposit function, when using standard ants, it was found that ANTS still tended to favor the generation of fairly dense networks. Altering the number of ant agents used to explore the structure as a means to control density of RNN candidates proved to help somewhat but was rather unwieldy and entailed far too much external human intervention. Instead, an ant agent role specialization scheme was developed, which was found to work far better as an automatic control mechanism to control the network size and synaptic density.

The first agent role, the *explorer ant*, can only choose from forward connections in the connectivity superstructure. The connections selected by this specialized agent are utilized to generate the base neural structure upon which recurrent connections are then be added to. After the explorer ants have selected the possible nodes and forward connections, two additional specializations of what is called *social ants* are then used: *i)*

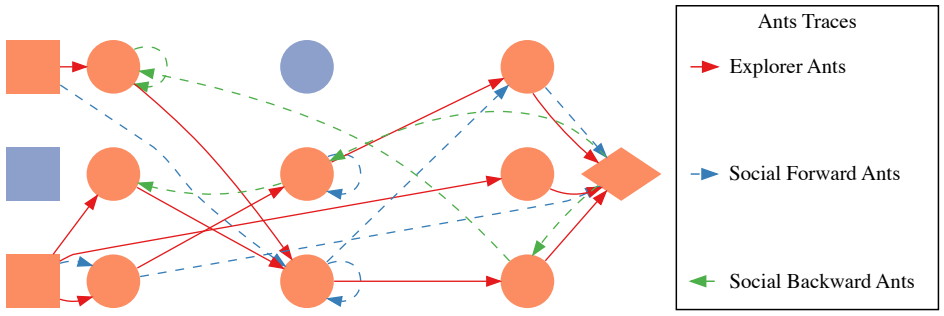


Figure 4.13: *In multi-role traversal, explorer ants (red) first select the forward paths in the network, creating a basic structure for the RNN. The social ant agents then select from the nodes chosen by the explorer ants. Within the social ant agent role, there is a sub-specialization consisting of forward recurrent ants (blue) that create additional forward recurrent connections between these nodes and backward recurrent ants (green) that move backwards from the output toward the input, creating backward recurrent connections between the same nodes.*

forward recurrent ants and, *ii) backward recurrent ants*. Social ants are first restricted to only visiting nodes that have already been selected by the explorer ants. In the case of the forward recurrent ants, when a path is chosen, the ant specifically creates a recurrent connection that moves forward in the network along the same path, along with a selected time skip (determined by pheromones). Backward recurrent ants, on the other hand, move backwards through the network and, for each path between nodes they take, a backward recurrent connection is added, along with a selected time skip (also determined by pheromones). Figure 4.13 provides an example of possible pathways that these specialized agents can take in a colony superstructure.

In addition to the development of specialized ant agents as described above, two modes for general ant movement were explored; *i)* ants were allowed to pick edges that could jump over layers in the colony (*i.e.*, the superstructure is massively connected, with a plethora of skip connections), or *ii)* ants were only allowed to select edges between consecutive layers (*i.e.*, the superstructure is fully connected, with no skip connections). This was tested to see the impact that layer skipping would have on the sparsity and performance of generated RNNs. Jumping and non-jumping modes were tested for both the standard ants (with and without forward-path bias) and the specialized ant agent roles.

4.2.4 Updating Pheromone Values

Different strategies for pheromone placement were also examined. τ is defined as the pheromone value, α as the pheromone decay parameter, W as the weights of the evaluated (candidate) RNN, and η as the candidate model's fitness. Specifically, four different functional schemes are described, which are used to model pheromone deposits.

The first implemented strategy for ANTS is standard for classical ACO setups. This deposit scheme rewards well performing RNNs with a fixed (constant) pheromone deposit while penalized ill-performing RNN models by evaporating the pheromone trace by a constant evaporation value, C. Specifically, this approach is defined as:

$$\tau_{new} = \tau_{old} \pm C \tag{4.9}$$

The second implemented strategy was one that used the fitness (value) as a parameter to guide the pheromone deposit. This has been shown to improve ACO performance in prior studies [131]. This scheme is defined as follows:

$$\tau_{new} = (1 - \alpha) \cdot \tau_{old} + \alpha \frac{1}{\eta} \tag{4.10}$$

The third strategy was to use the values of the neural synaptic weights themselves to control/guide the deposit of pheromones. Specifically, a penalty was exerted on the weights, specifically an L1 penalty (assuming a Laplacian prior of the synaptic weight values), in order to encourage regularization that favors sparser connectivity structure. This form of weight decay is sometimes applied to ANNs when controlling for over-parameterization and sparse weight matrices (with many near hard-zero values) are highly desirable. L1 regularization was applied to the pheromone deposition calculation in the following manner:

$$\tau_{new} = (1 - \alpha) \cdot \tau_{old} + \alpha \Big\{ \frac{1}{\eta + \frac{\gamma}{n} \|W\|} \Big\} \tag{4.11}$$

The fourth and final employed strategy was to insert an L2 penalty to regularize the RNN candidate weights. This assumes a Gaussian prior over the synaptic weight values and is sometimes referred to in ANN literature as "weight decay". L2 regularization is incorporated into pheromone deposition according to the following formula:

$$\tau_{new} = (1 - \alpha) \cdot \tau_{old} + \alpha \Big\{ \frac{1}{\eta + \frac{\gamma}{2n} \|W\|_2} \Big\} \tag{4.12}$$

These L1 and L2 functional variations of pheromone deposit schemes were developed in the hopes that they would ultimately encourage/reward the recovery of sparse, compact RNN predictive models.

4.2.5 Pheromone Evaporation

Lastly, pheromone trace values (deposited on the superstructures synaptic edge pathways) were allowed to evaporate or "decay" after each generation of an RNN in order to reduce the amount of pheromones on synaptic edges that have not been recently beneficial and to encourage exploration [92, 100, 131]. Pheromone values are updated (or decayed) according to the following equation:

$$\tau_{updated} = (1 - \beta) \cdot \tau_{current} + \beta \cdot \tau_{original} \tag{4.13}$$

where $\tau_{updated}$ is the pheromone value after the update, $\tau_{current}$ is the current pheromone value, $\tau_{original}$ is the original baseline pheromone value, and β is the pheromone evaporation rate. This function evaporates the pheromone back towards the original baseline value.

4.3 Using ACO to Optimize Neural Structure in a Continuous Domain

Although ANTS offered well performing RNN structures through an optimization process over a massively connected structure, the search space remains discrete, which represents a limitation to the search process. Further, as described previously, the size of the superstructure increases exponentially as additional layers or recurrent depths are added. Additionally, the maximum number of layers, nodes per layer and recurrent depth need to be pre-specified as hyperparameters to the algorithm, and poor selections can lead to poorly performing structures.

Continuous ANTS (CANTS) was developed as a potential solution to this problem, allowing networks of any size to be designed by an ACO process. However, adopting a search space that is continuous has different challenges such as how to consolidate the nodes from the scattered points picked by the optimization agents (continuous ants, or

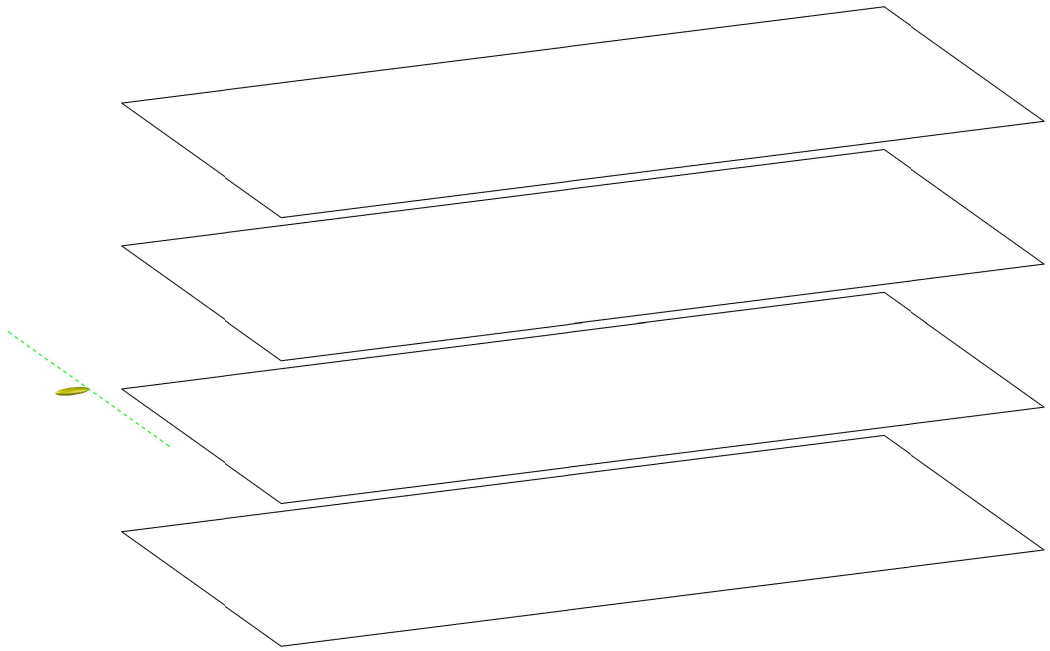
cants) in the search space, and how to set pheromones for those agents to communicate as a colony to optimize the paths they take to build the networks.

Similar to ANTS, the CANTS procedure (depicted in the the high-level pseudo-code in Algorithm 2) also employs an asynchronous, distributed "work-stealing" strategy to allow for scalable execution on HPC systems. The work generation process maintains a population of the best-found RNN architectures and repeatedly generates candidate RNNs whenever the worker processes request them. This strategy allows workers to complete the training of the generated RNNs at whatever speed they are capable of, yielding an algorithm that is naturally load-balanced. Unlike synchronous parallel evolutionary strategies, CANTS scales up to any number of available processors, supporting population sizes that are independent of processor availability. When the resulting fitness of candidate RNNs are reported to the work generator process, *i.e.*, mean squared error over validation data, if the candidate RNN is better than the worst RNN in the population, then the worst RNN is removed and the candidate is added. Note that the saved pheromone placement points for the candidate are incremented in the continuous search space.

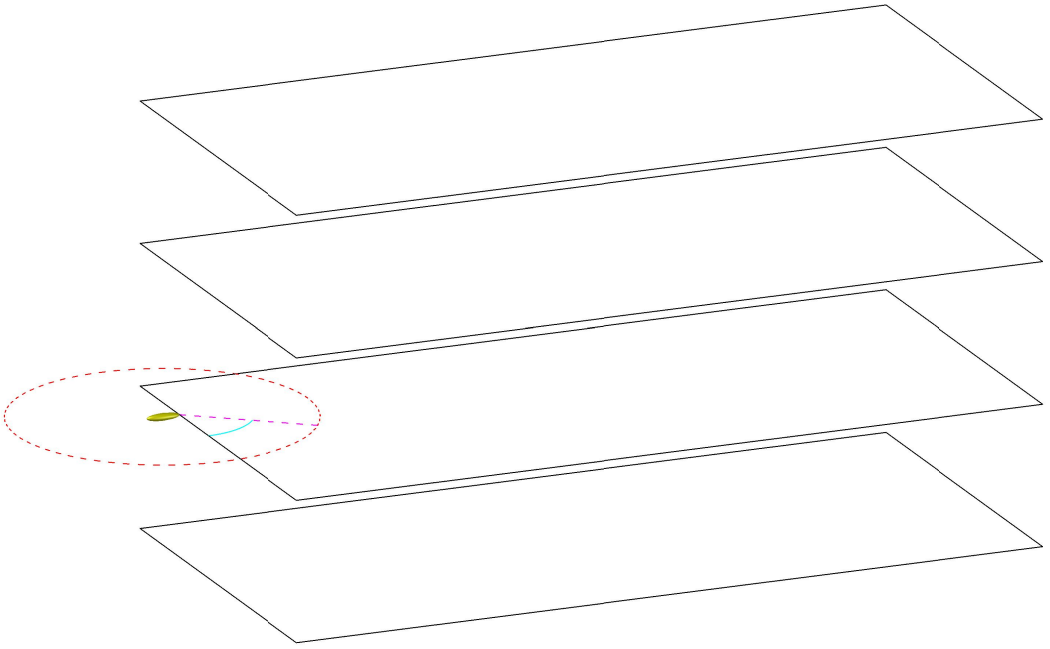
A group of solutions were designed to overcome the challenges discussed above. These solutions are as follows:

4.3.1 A Continuous Network Representation

RNNs are synthesized using a search space that could be likened to a stack of continuous 2D planes, where each 2D plane or slice of this stack represents a particular time step t (see Figure 4.14b). The input nodes for each time step are uniformly distributed at the input edge of the search space. A synthetic continuous ant agent (or *cant*) picks one of the discrete input node positions to start at and then moves through the continuous space based on the current density and distribution of other pheromone placements. Cants are allowed to move forward on the level they are on and can move up to any of the ones above it. However, they are restricted from moving down the stack – this constraint is imposed because the movement of ants between the layers in our algorithm's search space represents the forward propagation across time-steps, hence it is only possible to propagate information from a previous time step $(t - k)$ up to and including the current step t but not the reverse, since this would imply carrying unknown, future

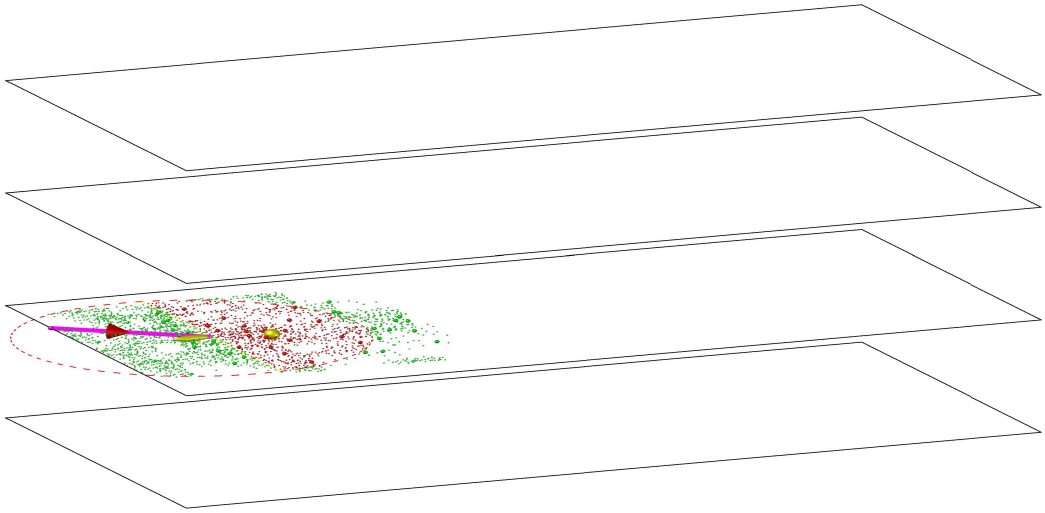


(a) *An ant (ovoid) starts by picking an input & a layer to start from.*

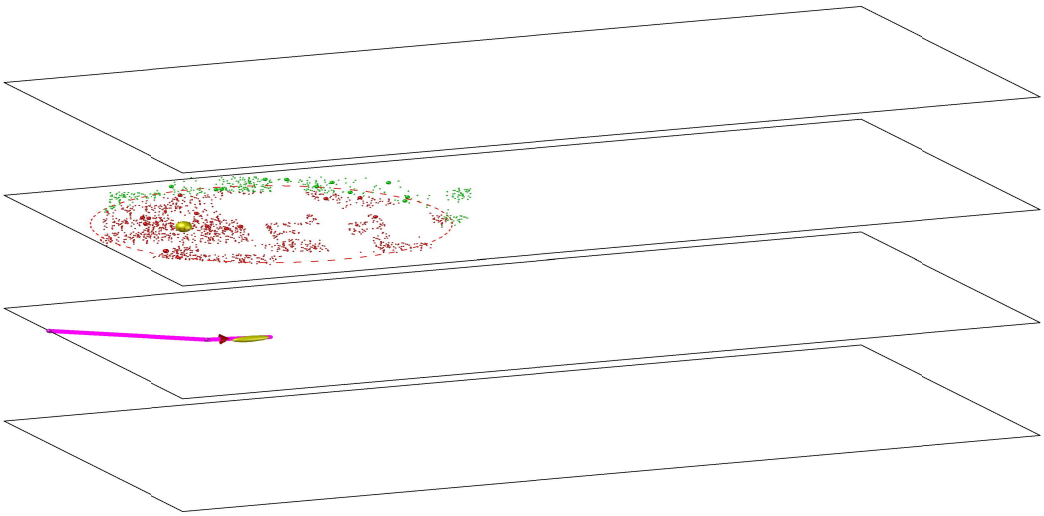


(b) *After an ant picks a layer to start with and an input node, it decides if it will create a new node (exploration), or follow pheromone traces (exploitation). If the former, the ant will randomly pick an angle between* $0°$ *and* $180°$*. It will use this angle to calculate the* x *and* y *components of its new position using a its sensing radius.*

Figure 4.14: CANTS' Movement.

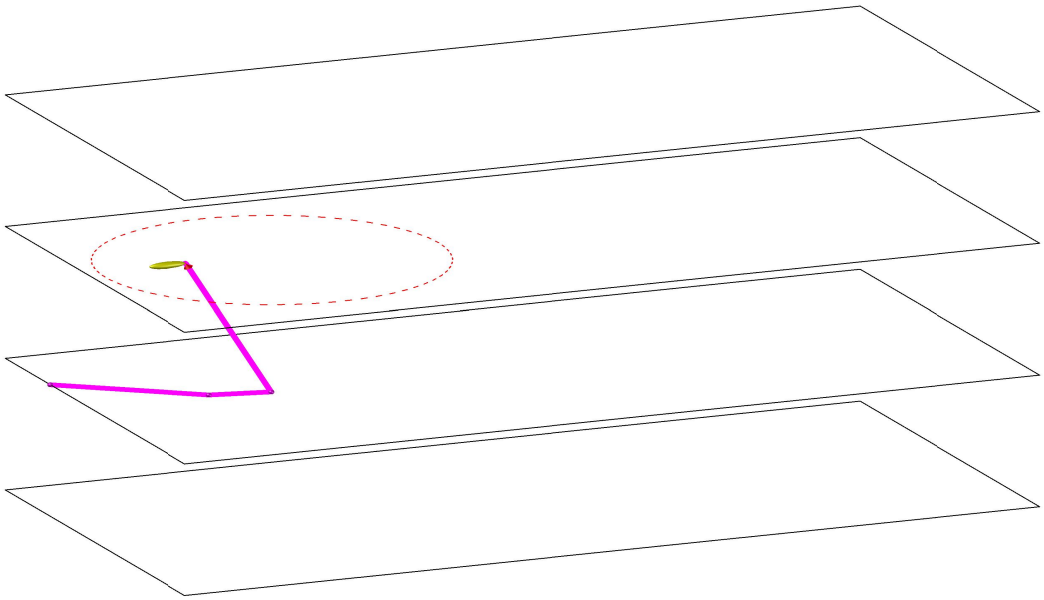


(c) *When the ant wants to use pheromone traces to determine its new point, it will first sense the pheromone traces within its sensing range (the sensing radius). The ant did not change its layer (the current position is on the same layer as the new position), the ant will only consider the pheromone traces between the angles* $0°$ *and* $180°$ *so not to go back on the same layer. The ant will then calculate the center of mass of the pheromone traces it is considering in its scanning and then move to that center of mass (sphere).*

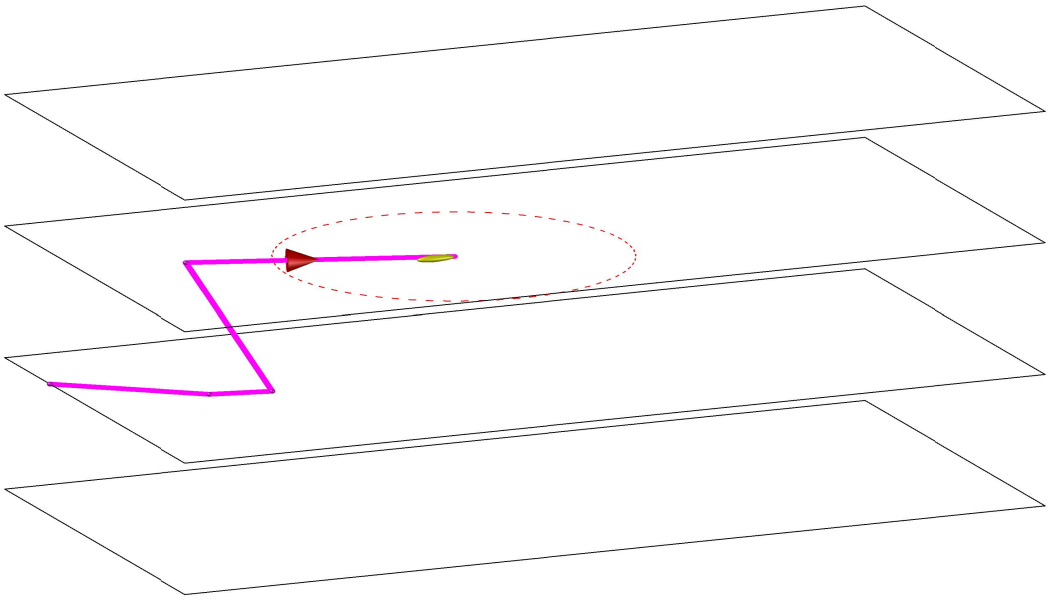


(d) *When the ant moves to an upper level from a lower level and it decides that it will use exploitation, it will consider the pheromone traces in its sensing range, which lies between the angles* $0°$ *and* $360°$*. This way, the ant can move backwards when jumping from a layer to another which makes a recurrent connection that goes back between hidden layers.*

Figure 4.14: CANTS' Movement (Continue).

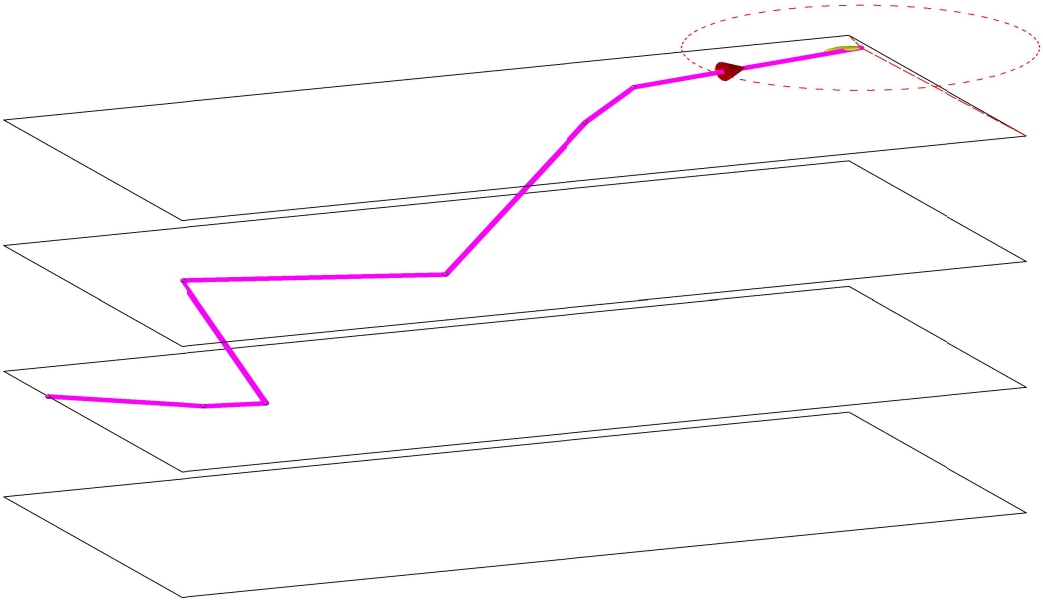


(e) *The ant will move to the new point and start a new move to reach to its destination (an output node)*



(f) *The ant will move to the new point and start a new move to reach to its destination (an output node)*

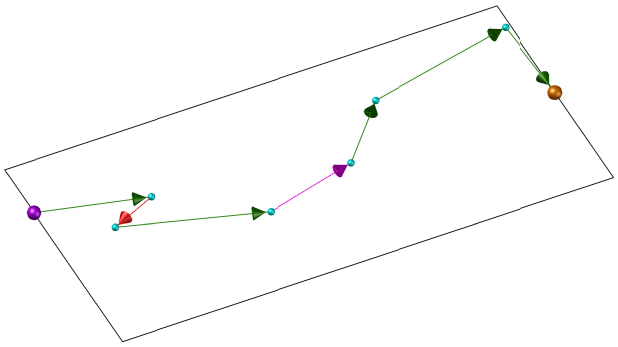
Figure 4.14: CANTS' Movement (Continue).



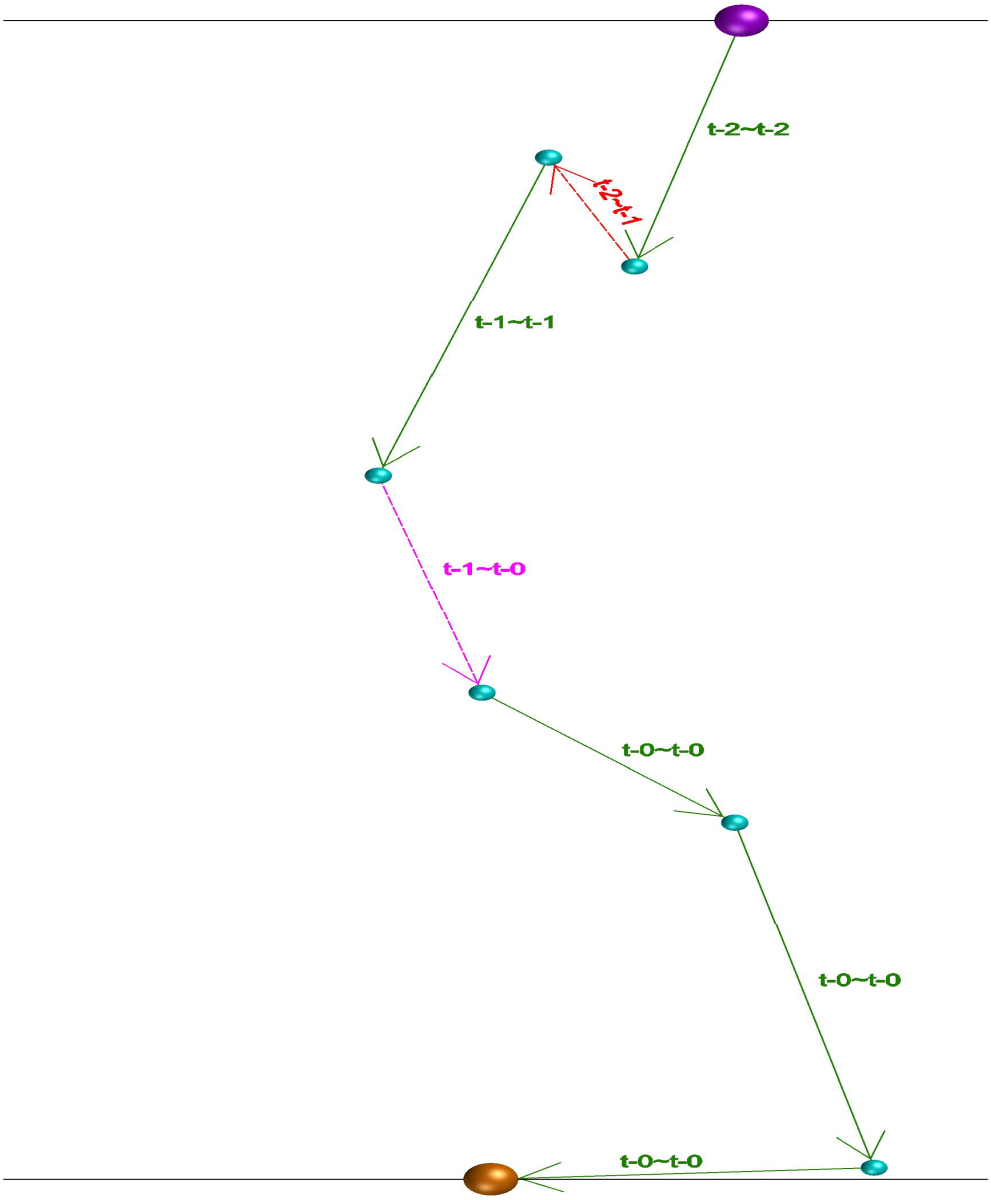
(g) *When the ant nears the output nodes, it will stop the continuous search and start picking an output node type based on a discrete search. If the set of the output nodes is only one, then the ant will directly connect its latest point to the output. The points of the path of the ant is translated to nodes, edge*

Figure 4.14: CANTS' Movement (continued).

signals backwards. While ants only move forward on a given plane, they are permitted to move backward on a slice if they had just moved to it form a lower level (since many RNNs have synapses that potentially skip neuronal layers). This enforced upward and (overall) forward movement ensures that cants continue to progress towards outputs and do not needlessly circle around in the search space. Figures 4.14 shows examples of how cants move from an input edge of the search space to its output edge, how cants explore new regions in the search space, how cants exploit previously searched areas via attraction to deposited pheromones, and how cants' path through the space are translated into a final, candidate RNN. The software developed that implements our CANTS procedure also provides a replay visualization tool so that traces from a run can be visualized to see how pheromones are deposited and how RNNs are generated, as shown in Figure 4.14.

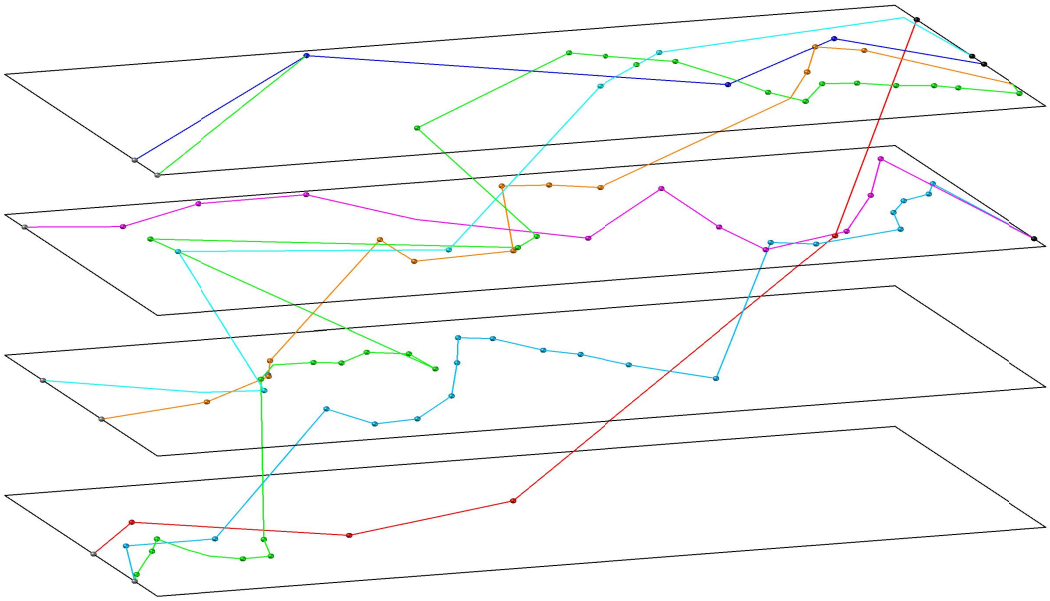


(h) *Ant's path is projected on one plane*

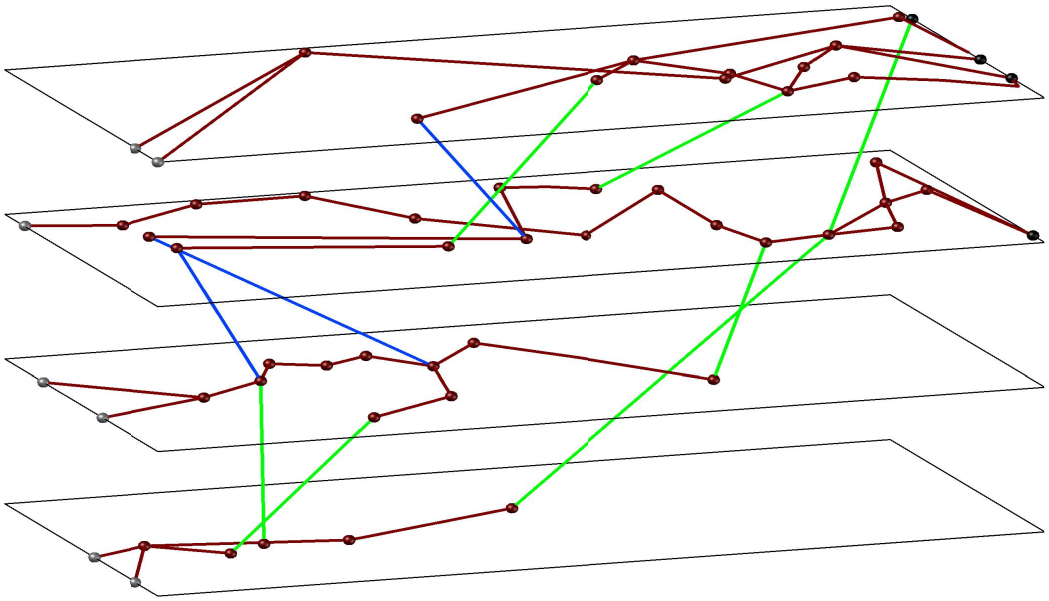


(i) *The ant picked its input point, started at t_{-2}, picked a node at t_{-2} (edge), picked a node at t_{-1} (backward recurrent edge), picked a node at t_{-1} (edge), picked a node at t_0 (forward recurrent edge), picked a node at t_0 (edge), picked a node at t_0 (edge), and finally picked an output node at t_0 (edge)*

Figure 4.14: CANTS' Movement (continued).

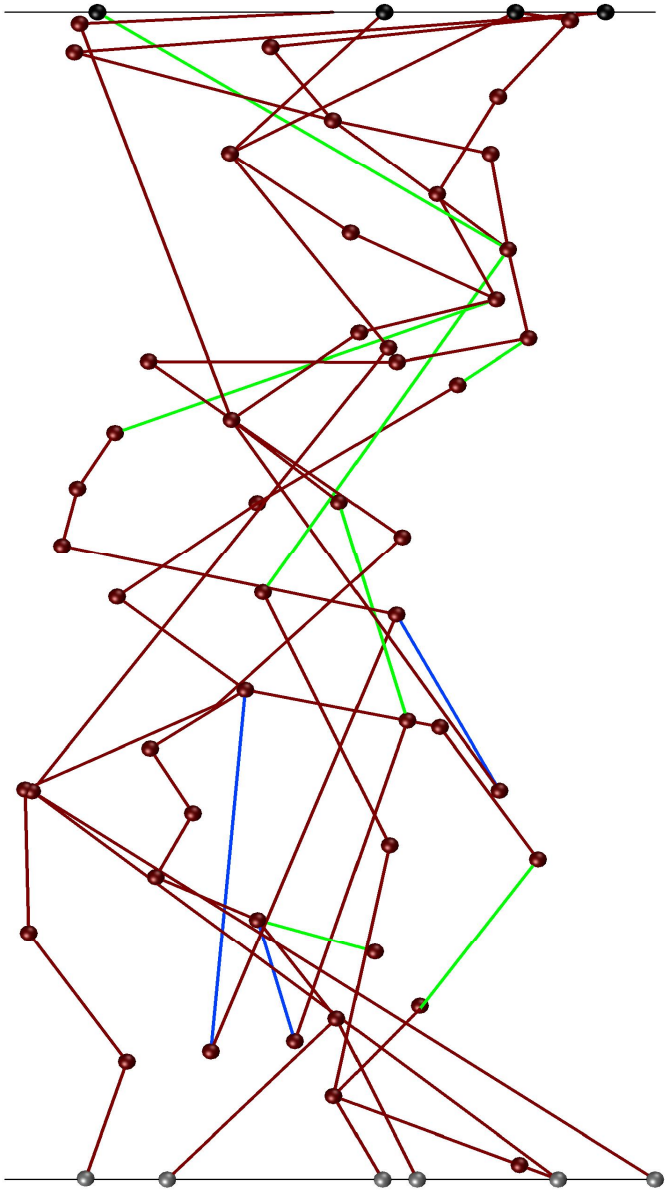


(j) *Several ants make their path from an input to an output*



(k) *The ants' nodes on each level is then condensed (clustered) based on their density.*

Figure 4.14: CANTS' Movement (continued).



(I) *The final network is the final result of clustering the node and defining the connections between nodes in the same layer as edges, and the connection between nodes and between layers as forward recurrent edges or backward recurrent edges. The flow goes from the inputs at the bottom to the outputs at the top.*

4.3.2 Cant Agent Input Node and Layer Selection

Each level in the search space has a level-selection pheromone value, p_l, where l is the level. These are initialized to:

$$p_l = 2 * l \tag{4.14}$$

where the top level for the current time step is $l = 1$, the next level for the first time lag is $l = 2$ and so on. A cant selects its starting level according to the probability of starting at level l as:

$$P(l) = \frac{p_l}{\Sigma_{l=1}^{L} p_l} \tag{4.15}$$

where L is the total number of levels. This scheme encourages cants to start at lower levels of the stack at the beginning of the search. After selecting a level, the cant selects its input node in a similar fashion, based on the pheromones for each input node location on that level. When a candidate RNN is inserted into the population, the amount of pheromone level on the points in the search space, corresponding to the nodes that were utilized by that RNN, is incremented.

4.3.3 Cant Agent Movement

To balance exploration with exploitation, cants behave similarly to real-world ants by following communication clues to reach to targets. When a cant moves, it first decides if it will climb up to a higher (stack) level. This is done in the same manner as selecting its initial layer, except that it only selects between its current level and higher ones. After deciding if it will climb or not, the agent will then decide if it will explore or exploit. Cants randomly choose to exploit at a percentage equal to exploitation parameter ϵ.

When a cant decides to exploit and follow pheromone traces, *i.e.*, clues, it will start sensing the pheromone points around it, given a sensing radius, ρ. If the cant is staying on the same level, it will only consider deposited pheromones that are in front of it (*i.e.*, closer to the output nodes), otherwise, it will consider all the pheromones that are inside its sensing radius on the level it is moving to. The cant then calculates the center of mass of the pheromones in this region using the point in the space it will move to. This point is then saved by the candidate RNN (as a point to potentially

increment pheromone values) if the RNN is later to be inserted into the RNN population. Since cants consider the center of mass of the pheromone values, the individual points of pheromone values are not the effective factor in the cant-to-cant communication. Rather, it is the concentration of the pheromone in a region of the space that more closely aligns with how real ants move in nature.

When a cant decides that it will explore, it instead selects a random point that lies within the range of their sensing radius to move to. Once a cant decides if it is climbing or staying in the same level, it will generate an angle bisector that is either a random number between $[0, 1]$ if the current and next point are on the same level or $[-1, 1]$ if the current and next points are on different levels. This angle bisector is used to calculate the angle of the next movement of the cant:

$$\theta = angle\_bisect * PI \tag{4.16}$$

The movement angle is then subsequently used to calculate the next x and y coordinates of the next position of the cant:

$$x_{new} \leftarrow x_{old} + \rho * cos(\theta) \tag{4.17}$$

$$y_{new} \leftarrow y_{old} + \rho * sin(\theta) \tag{4.18}$$

These points are also saved for potentially future pheromone modification.

4.3.4 Condensing Ants Segments' Points

Cants choose the points in their paths from the inputs to the outputs, the points in the search space are clustered using the density-based spatial clustering of applications with noise [48] (DBSCAN) algorithm to condense those points to centroids. The points of the segments of the cants' paths are then shifted to the centroids that they belong to in the search space and those new points become the nodes of the generated RNN architecture (see Figures 4.14j and 4.14k). The node types are picked by a pheromone-based discrete local search, as done in the discrete space ANTS. Each of these node types at the selected point will have their own pheromone values which drive probabilistic selection.

4.3.5 Communal Weight Sharing

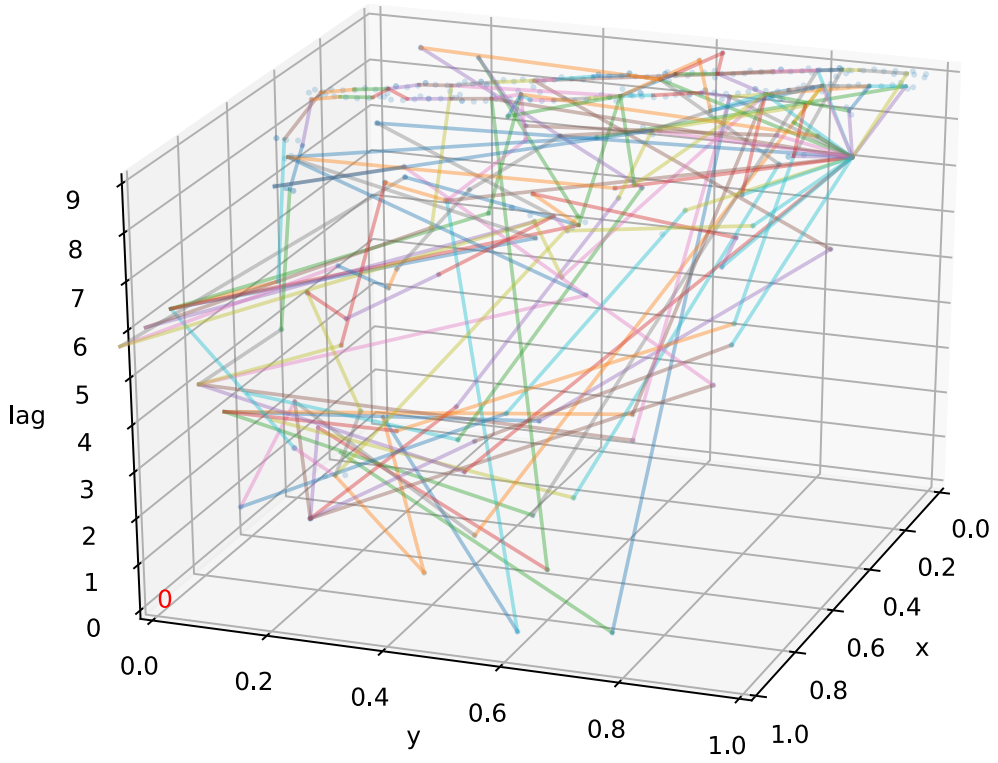
In order to avoid having to retrain every newly generated RNN from scratch, a communal weight sharing method has been implemented to allow generated RNNs to start with values similar to those of previously generated and trained RNNs. The centroid points (*i.e.*, the RNN node points in the continuous space) in CANTS retain the weights of all the out-going edges from those nodes. Each newly-created centroid is assigned a weight value which is passed to the edges of the generated RNN. In case where a centroid did not have any previously created centroid in its cluster, randomly initialized weights are assigned to those outgoing edges either uniformly at random between -0.5 and 0.5, or via the Kaiming [62] or Xavier [57] strategies. If there were previously-created centroids in the clustering region, the weight values assigned to the generated RNN nodes are the average of the weights of those existing centroids. The weights of a centroid are updated after an RNN is trained by calculating the averages of the original centroid weight values and all the weights of the outgoing edges of the corresponding node (after training). The updated weights can then be used to initialize new centroid weights when they lie in their cluster when DBSCAN is applied in the following iteration.

4.3.6 Pheromone Points Volatility

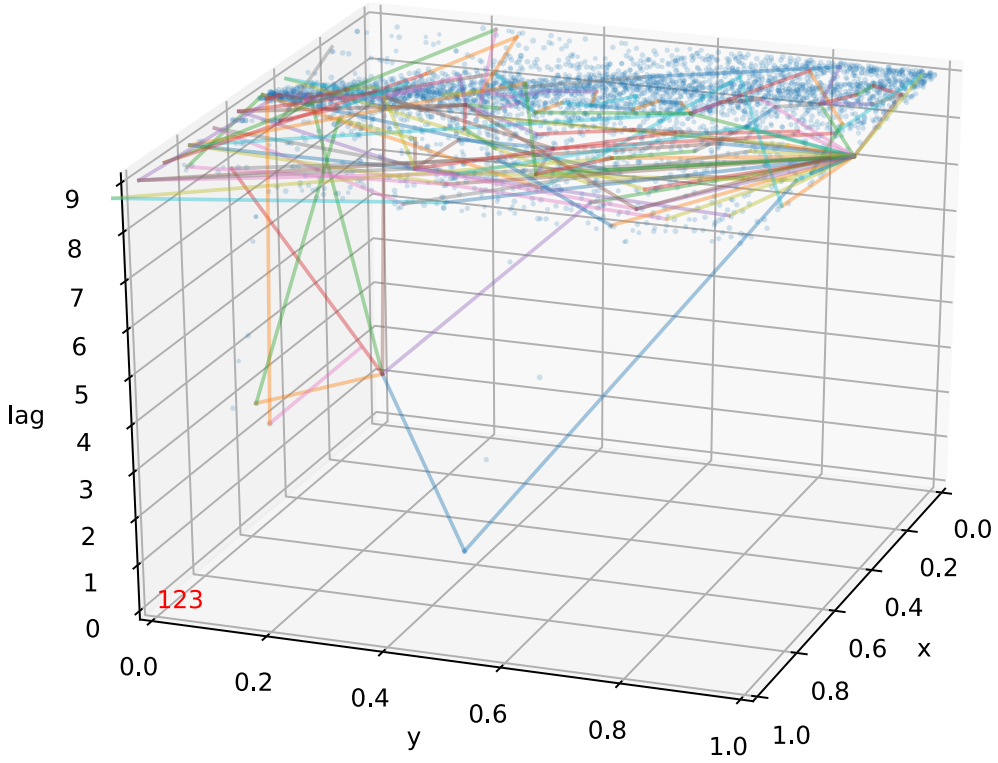
Pheromone decay happens on a regular basis after each iteration of optimization regardless of the performance of the generated RNN(s). The pheromones decay by a constant value and after a specific minimum threshold the point is removed from the search space. By letting points vanish, the search space removes tiny residual pheromones which might provide distraction to cant-to-cant communication as well as slow down the overall algorithm.

4.3.7 Pheromone Placement

For each successful candidate RNN, *i.e.*, each RNN that performs at least better than the worst in the population, the corresponding centroids for its RNN nodes in the search space are rewarded by increasing their pheromone values by a constant value. The values of the pheromones have a maximum limit to avoid becoming overly attractive points to the cants, which could result in premature convergence.



(a) Frame a



(b) Frame b

Figure 4.14: *CANTS in Action*

Algorithm 1 Ant-guided Neural Topology Search Algorithm

procedure MASTER

▷ construct fully connected structure with edges holding initial pheromone and weights values.

$colony$ = **new** $Colony$

for $i \leftarrow 1 \ldots max\_iteration$ **do**

$nn_{new} \leftarrow ants\_swarm(colony)$

$send\_to\_worker(nn_{new}, worker.id)$

$nn_{new}, fit \leftarrow receive\_fit\_from\_worker()$

if $nn\_fitness < worst\_population\_member$ **then**

$population.pop(worst\_population\_member)$

$population.add(new\_nn)$

$reward\_paths\_in\_colony(new\_nn)$

if $use\_Lamarckian\_weight\_inheritance$ **then**

if $use\_phi\_function$ **then**

▷ update colony's weights from nn_{new} using phi equation

else if $constant\_phi$ **then**

▷ update colony's weights with constant fraction of nn_{new} weights

if $bias\_forward\_paths$ **then**

▷ Sum (Fwd Edge/Recurrent Pheromone) = Sum(Bkwd Recurrent Edges Pheromone)

$periodic\_pheromone\_evaporation()$

procedure $Reward\_Paths(nn)$

for each $edge \in nn.edges$ **do**

if $use\_constant\_reward$ **then**

$colony.edge[edge.id].pheromone$ += $constant$

else if $use\_fitness$ **then**

$colony.edge[edge.id].pheromone$ = Eqn 4.10

else if $use\_L1\_regularization$ **then**

$colony.edge[edge.id].pheromone$ = Eqn 4.11

else if $use\_L2\_regularization$ **then**

$colony.edge[edge.id].pheromone$ = Eqn 4.12

```
procedure WORKER
    recieve_from_master(nn)
    fitness ← train_test_nn(nn)
    send_fitness_to_master(nn, fitness)
procedure Ants_Swarm
    if one_layer_jump then
        ▷ ants only move one layer at a step
    else if !one_layer_jump then
        ▷ ants can jump over layers
    if one_ant_species then
        for ant ← 1 . . . no_ants do
            ▷ ant chooses the nodes, edges, and recurrent edges
    else if two_ants_species then
        for ant ← 1 . . . no_ants/2 do
            ▷ ant choose the nodes, edges from colony
        if social_forward & !social_backward then
            for ant ← 1 . . . no_ants/2 do
                ▷ ants choose rec_edges only from fwd rec_edges
        else if social_backward & !social_forward then
            for ant ← 1 . . . no_ants/2 do
                ▷ ants choose rec_edges only from bwd rec_edges
        else if social_forward & social_backward then
            for ant ← 1 . . . no_ants/4 do
                ▷ ants choose rec_edges only from fwd rec_edges
            for ant ← 1 . . . no_ants/4 do
                ▷ ants choose rec_edges only from bwd rec_edges
     return new_nn
```

Algorithm 2 Continuous Ant-guided Neural Topology Search Algorithm

procedure $Maestro$
▷ construct search space with inputs at y=0 and output at y=1
▷ recurrency time-lag steps is the spaces's z axis
$search\_space =$ **new** $SearchSpace$
for $i \leftarrow 1 \dots max\_iteration$ **do**
$nn_{new} \leftarrow AntsSwarm()$
$send\_to\_worker(nn_{new}, worker.id)$
$nn_{new}, fit \leftarrow receive\_fit\_from\_worker()$
if $nn\_fitness < worst\_population\_member$ **then**
$population.pop(worst\_population\_member)$
$population.add(nn_{new})$
$RewardPoints(nn_{new})$

procedure $Worker$
$receive\_from\_master(nn)$
$fitness \leftarrow train\_test\_nn(nn)$
$send\_fitness\_to\_master(nn, fitness)$

procedure $AntsSwarm$
▷ Ants choose input in discrete fashion
for $ant \leftarrow 1 \dots no\_ants$ **do**
$CreatePath(ant)$
▷ Use DBscan to cluster ants paths points
$segments \leftarrow DBscanPaths(ants)$
▷ Create RNN from segments
$rnn_{new} \leftarrow CreateRNN(segments)$ **return** rnn_{new}

procedure $CreatePath(ant)$
▷ Choose input in discrete fashion
$ChooseInput(ant)$
▷ Create a path starting from the input
while $ant.current\_y < 0.99$ **do**
$r \leftarrow$ **uniform_random**$(0, pheromone\_sum - 1)$
$ant.current\_level \leftarrow ant.climb$
if $r > ant.exploration\_instinct$ or $search\_space[ant.current\_level]$ is not $Empty$ **then**
$point \leftarrow CreateNewPoint(ant.search\_radius)$
$ant.path.insert(point)$
$search\_space.insert(point)$
else
$point \leftarrow FindCenterOfMass(ant.current\_position, ant.search\_radius)$
if $point$ not in $search\_space[ant.level]$ **then**
$ant\_path.insert(point)$
▷ Choose Output in discrete fashion
$ChooseOutput(ant)$

```
procedure ChooseInput(ant)
    ▷ select input probabilistically according to pheromones
    pheromone_sum ← sum(pheromones.input)
    r ← uniform_random(0, pheromone_sum − 1)
    ant.input ← 0
    while r > 0 do:
        if r < pheromones.input[ant.input] then
            ant.input ← 1
            break
        else
            r ← r − pheromones.input[ant.input]
            ant.input ← ant.input + 1
procedure ChooseOutput(ant)
    ▷ select input probabilistically according to pheromones
    pheromone_sum ← sum(pheromones.output)
    r ← uniform_random(0, pheromone_sum − 1)
    ant.input ← 0
    while r > 0 do:
        if r < pheromones.input[ant.output] then
            ant.output ← 1
            break
        else
            r ← r − pheromones.output[ant.output]
            ant.output ← ant.output + 1
procedure DBscanPaths(ants)
    for ant ← 1 . . . num_ants do
        for point ← 1 . . . ant_path do
            segments[ant].insert(PickPoint(point))
    return segments
procedure PickPoint(point)
    [node, points_cluster] ← DBscane(point, search_space[point.level])
    node.out_edges_weights.insert(AvrgWeights(points_cluster))
    search_space.insert(node) return node
procedure RewardPoints(rnn)
    for each node ∈ rnn.nodes do
        search_space[node].pheromone += constant
        search_space[node].weight ← average_weight(node.weight, search_space[node].weight)
        if search_space[node].pheromone > PHEROMONE_THRESHOLD then
            search_space[node].pheromone = PHEROMONE_THRESHOLD
```

Chapter 5

Results

In this chapter, we present the experiments using MC-ACO for optimizing an RNN sub-structure succeeded in not only improving the performance, but also in providing sparser and lighter structures as discussed in Section 5.2. The experiments used civil-aviation engine time-series data collected from FDRs to predict severe engine vibrations. Section 5.3 presents the results obtained by applying ANTS and its heuristics. The experiment involved coal fired power plant operation time-series data to predict the main flame intensity of the burner of the plant. Finally, Section 5.4 presents the results of using a continuous search space in CANTS instead of a discrete one when using ACO for NAS. The time-series datasets used belong to the coal power generation industry and the wind power generation industry. They were used to predict the net plant heat rate and main flame intensity of the coal fired power plant, and the output power of the wind power generators. The results of of ANTS and CANTS are compared to each other and to EXAMM on the same datasets.

Computing Environments The results for MC-ACO were obtained using the University of North Dakota's high performance computing cluster. The cluster is running the Red Hat Enterprise Linux (RHEL) 7.2 operating system with 31 nodes, each with 8 cores for 248 in total, 64GBs RAM per node for a total 1948 GB, and it is using InfiniBand 10 gigabit (GB) for interconnect.

The results for ANTS and CANTS were obtained by scheduling the experiment on

RIT's high performance computing cluster with 64 Intel® Xeon® Gold 6150 CPUs, each with 36 cores and 375 GB RAM (total 2304 cores and 24 TB of RAM). Each ANTS experiment utilized 15 nodes (540 CPU's), taking approximately 30 days to complete all the experiments. CANTS experiments used 5 nodes (180 CPU's), taking 7 days to finish three experiments.

Error Functions For all the networks studied in this work, Mean Squared Error (MSE) (shown in Equation 5.1) was used as an error measure for training, as it provides a smoother optimization surface for backpropagation than mean average error. Mean Absolute Error (MAE) (shown in Equation 5.2) was used as a final measure of accuracy for the three architectures, as because the parameters were normalized between 0 and 1, the MAE is also the percentage error.

$$Error = \frac{0.5 \times \sum(Actual\ Vib - Predicted\ Vib)^2}{Testing\ Seconds} \tag{5.1}$$

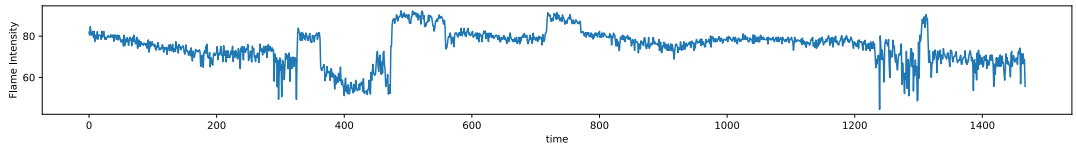
$$Error = \frac{\sum[ABS(Actual\ Vib - Predicted\ Vib)]}{Testing\ Seconds} \tag{5.2}$$

5.1 Dataset Descriptions and Details

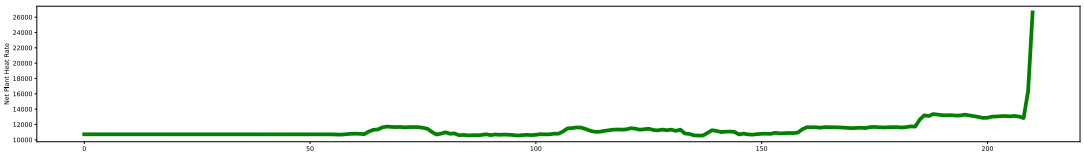
The data[1] used in the experiments are time-series data that belong to the power industry. Two of the datasets are obtained from a coal fired power plant and they are logged from the operations of:

- **The plant's burner:** This data set has 12 parameters, which were used to predict the 'Main Flame Intensity' of the burner one time unit in the future. An example of this parameter's data is presented in Figure 5.1a.
- **The plant's boiler:** This data set has 48 parameters, which were used to predict the 'Net Plant Heat Rate' one time unit in the future. An example of this parameter's data is presented in Figure 5.1b

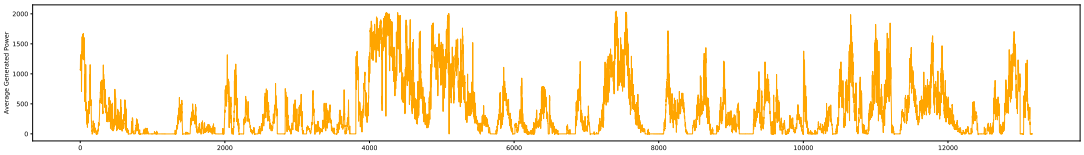
[1] All used datasets are publically available on https://github.com/travisdesell/exact/tree/main/datasets



(a) *Flame Intensity in Coal Fired Power Plant's Burner*



(b) *Net Plant Heat Rate in Coal Fired Power Plant's Boiler*



(c) *Wind Turbine Average Generated Power*

Figure 5.1: *Examples of the time series data for the predicted parameters in the different data sets used in this work.*

The third dataset belongs to the wind power generation industry. The data set contains 88 parameters, which were used to predict the 'Average Generated Power' of the wind power generator one time unit in the future. An example of this parameter's data is presented in Figure 5.1c. Table 5.1 depicts the datasets used, the numbers of their parameters, and their sizes.

	Flame Intensity	**Net Plant Heat Rate**	**Average Generated Power**
Number of Inputs	12	48	88
Training Records	7200	850	190,974
Testing Records	7200	211	37,514

Table 5.1: Numbers of parameters and recordings used from the different data sets for training and testing data.

5.2 MC-ACO Results

The experiments introduced in this section represents the first steps in investigating ACO for NAS. Several fixed architectures were used to test the effect of the size of the structure on the performance of the models. The devised LSTM cells are modified designs to fit the problem's input and the used architectures dimensional reductions between the layers. The LSTM models predicted severe aviation engine vibrations for 1, 5, 10, and 20 seconds in the future and the models were trained for 575 backpropagation epochs. The results are compared to the regression-based models NOE, NARX, and NBJ.

5.2.1 Programming Language

Python's Theano Library [146] was used to implement the artificial neural network models. It was chosen due to four major advantages: *i)* it will compile the most, if not all, of functions coded using it to C and CUDA providing fast performance, *ii)* it will perform the weights updates for backpropagation with minimal overhead, *iii)* Theano can compute the gradients of the error (cost function output) with respect to the weights, saving significant effort and time needed to manually derive the gradients, coding and debugging them (which is particularly challenging in regards to LSTM neurons), and finally, *iv)* it can utilize GPUs for further increased performance. The MC-ACO algorithm was implemented in Python using MPI for Python [23] to allow for it to be distributed on high performance computing environments.

5.2.2 Data Processing

The flight data parameters used were normalized between 0 and 1. The sigmoid function was used as an activation function over all the gates and inputs/outputs. The ArcTan activation function was tested on the data, however it gave distorted results and sigmoid function provided significantly better performance.

5.2.3 Comparison to Traditional Methods

As mentioned in Chapter 2, the NOE, NARX, and NBJ models were implemented as baseline comparison methods. These traditional models are dynamical systems can experience limitations which reduce their stability and ability to make most effective use of embedded memory. In particular, they can suffer from vanishing and exploding gradients [65, 114], especially when using the back propagation through time algorithm [154] on long time series such as the vibration data used in this work.

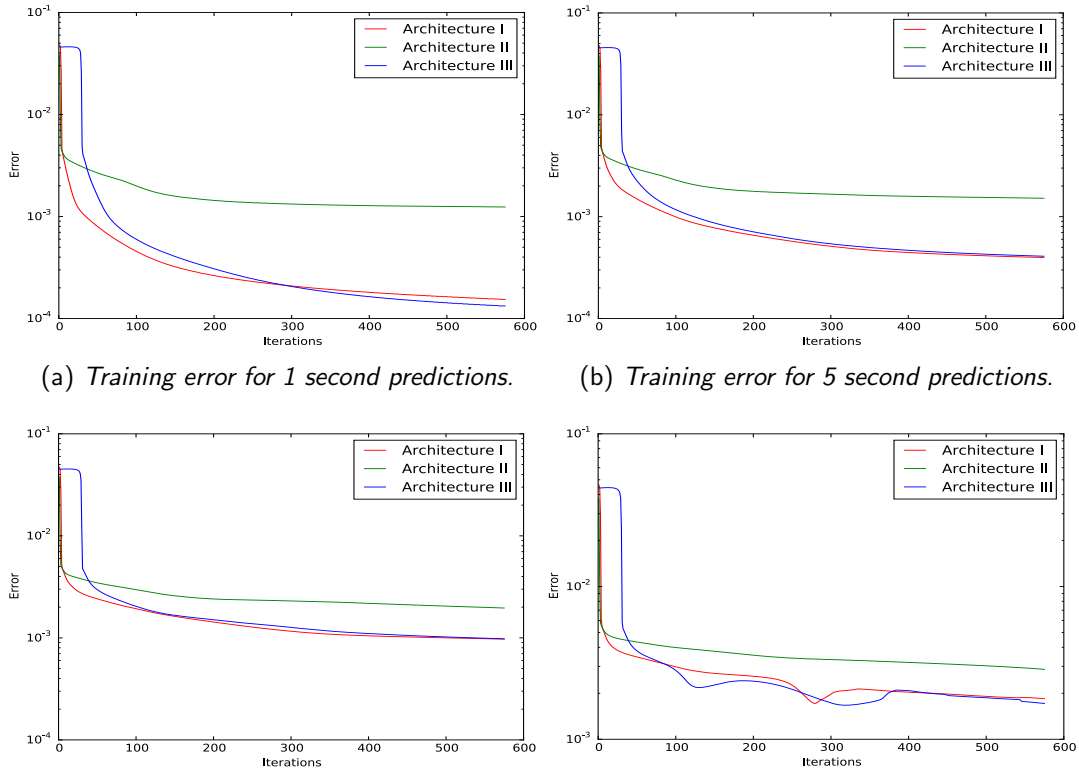
It should further be noted that the purpose of the work in this chapter is predict values multiple time steps into the future, which is not possible for the NBJ model, as it the actual value to be predicted along with the error between the prediction at that value to be fed back into the RNN at the next iteration. If this model is being used online to predict data 5, 10 or 20 seconds in the future, the output and error values will not be known for an additional 5, 10 or 20 time steps (given readings every second) until that time actually occurs. However, as the data used in this study has already been collected, we still evaluated these models in an offline manner where this future knowledge can be known for sake of comparison.

5.2.4 K-Fold Cross Validation

The 57 flights in the data were divided into 10 groups. Seven groups consisted of 6 flights, and the other three consisted of 5 flights. The groups where used to cross validate the results by running nine of the groups as training data set and use the tenth as a testing set. Then, one of the groups in the training set switch places with the group in the testing set and then another run is executed. By doing so, the study had ten runs to cross validate the results and compare them statistically.

5.2.5 Fixed Architecture Results

For Architectures I, II and III, the MSE for predicting 1 sec, 5 sec, 10 sec and, 20 sec during the training process is shown in Figure 5.2. Results are shown in logarithmic scale. Initially, all three architectures were trained for 575 epochs; however, additional epochs were later examined for Architecture III. In these preliminary results the vibra-



(a) *Training error for 1 second predictions.* (b) *Training error for 5 second predictions.*

(c) *Training error for 10 second predictions.* (d) *Training error for 20 second predictions.*

Figure 5.2: *Mean squared error during the training process for the three architectures predicting vibration in 1, 5, 10, and 20 seconds in the future.*

tion dataset was split into 28 flights in the training set, with a total of 41,431 seconds of data, and flights in the testing set, with a total of 38,126 seconds of data.

The final mean squared error on the training data is shown in Table 5.2. Figure 5.3 presents the predictions for a selection of flights from the test set, and Figures 5.4 provides an uncompressed example of predicting vibration 1, 5, 10 and, 20 seconds in the future over a single flight from the testing data. In the multiple flight plots, each flight ends when the vibration reaches the max critical value (normalized to 1) and after which the next flight in the test set begins.

Table 5.2: Mean Squared Error of the architectures' Training phase in previous study

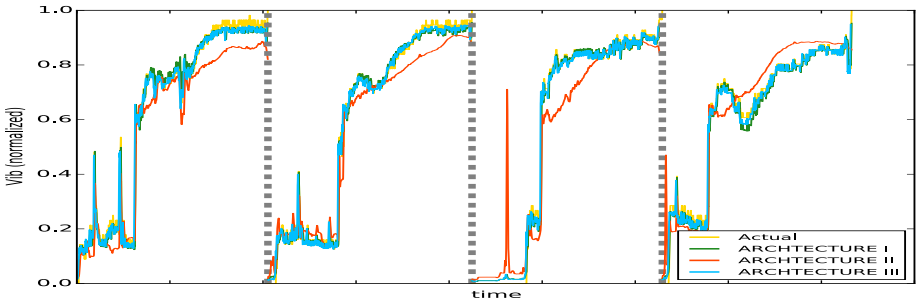
	Prediction Error (MSE)			
	1 seconds	**5 seconds**	**10 seconds**	**20 seconds**
Architecture I	0.000154	0.000398	0.000972	0.001843
Architecture II	0.001239	0.001516	0.001962	0.002870
Architecture III	0.000133	0.000409	0.000979	0.001717

Table 5.3: Mean Squared Error of the architectures' testing phase in previous study

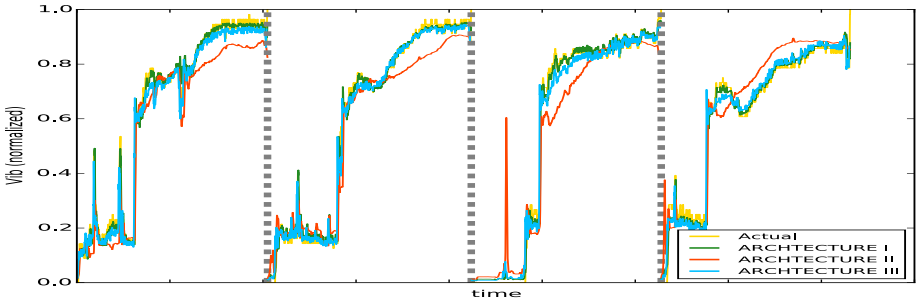
	Prediction Error (MSE)			
	1 seconds	**5 seconds**	**10 seconds**	**20 seconds**
Architecture I	0.000792	0.001165	0.002926	0.010427
Architecture II	0.010311	0.009708	0.009056	0.012560
Architecture III	0.000838	0.002386	0.004780	0.041417

Table 5.4: Mean Absolute Error of the architectures' testing phase in previous study

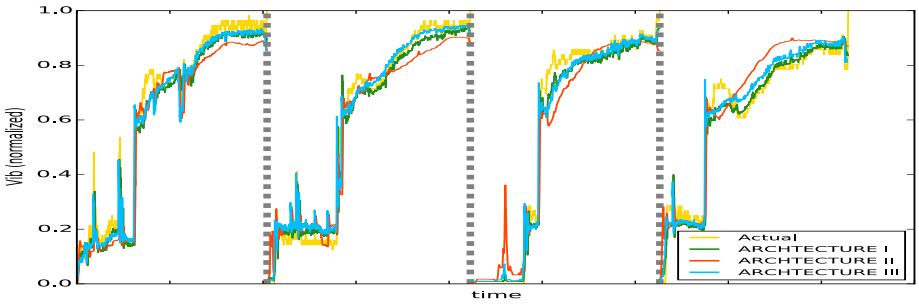
	Prediction Error (MAE)			
	1 seconds	**5 seconds**	**10 seconds**	**20 seconds**
Architecture I	0.028407	0.033048	0.055124	0.101991
Architecture II	0.098357	0.097588	0.096054	0.112320
Architecture III	0.027621	0.048056	0.070360	0.202609



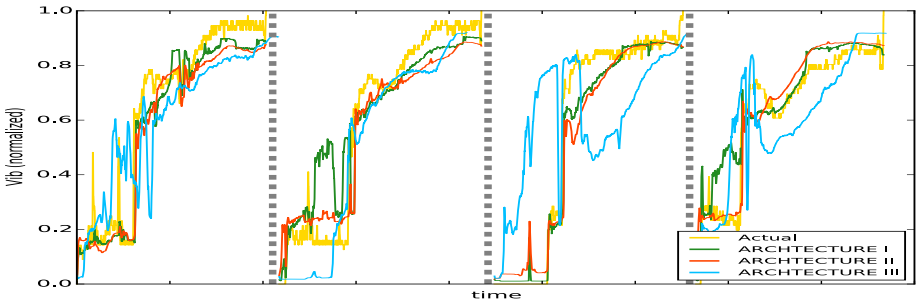
(a) *Predictions for 1 second in the future.*



(b) *Predictions for 5 seconds in the future.*

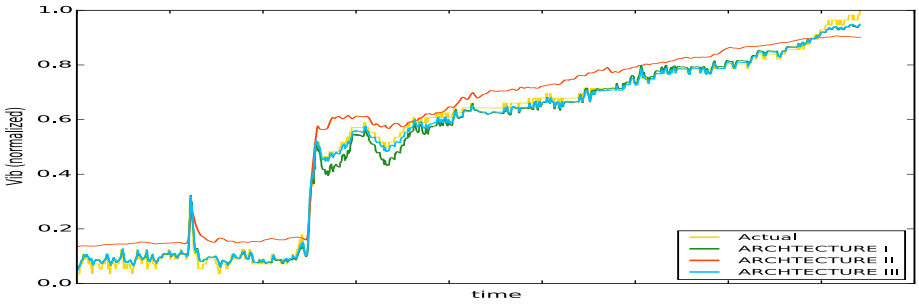


(c) *Predictions for 10 seconds in the future.*

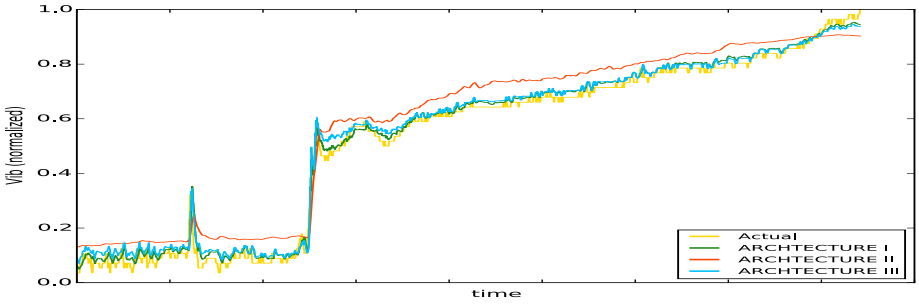


(d) *Predictions for 20 seconds in the future.s*

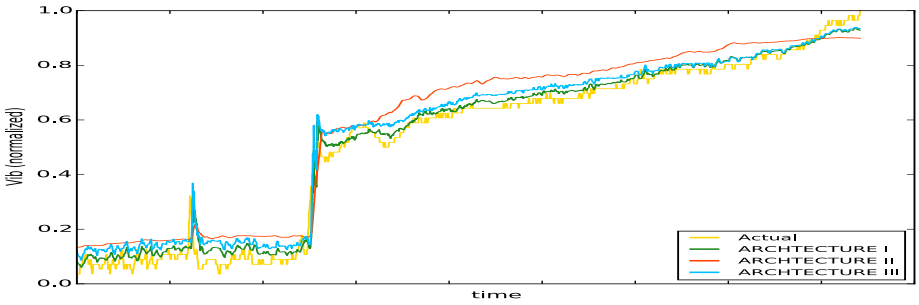
Figure 5.3: *Plotted results for Architectures I, II, and III for 1, 5, 10 and 20 seconds in the future for a selection of test flights.*



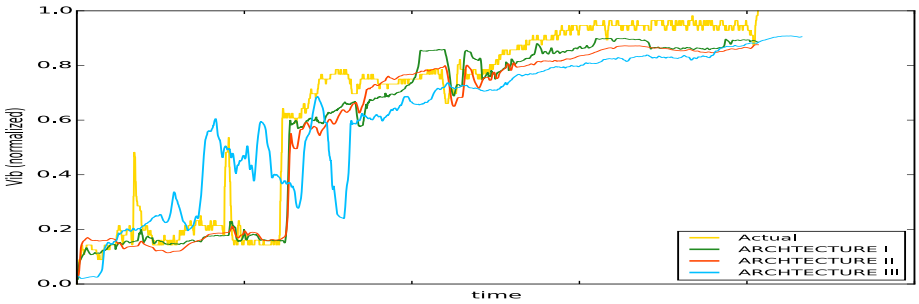
(a) *Predictions for 1 second in the future.*



(b) *Predictions for 5 seconds in the future.*



(c) *Predictions for 10 seconds in the future.*



(d) *Predictions for 20 seconds in the future.*

Figure 5.4: *Plotted results for Architectures I, II, and III for 1, 5, 10 and 20 seconds in the future for a single test flight.*

Results of Architecture I

The results of this architecture, shown in Table 5.3, came out to be the most accurate in predicting the vibration parameter. There is more misalignment between the actual and calculated vibration values as predictions are made further in the future, as shown in Figures 5.3, and 5.4, which present predictions over a selection of test flights and over a single flight in higher resolution, respectively. This misalignment is to be expected as it is more challenging to predict further in the future. Also, the results show that the prediction of higher peaks is more accurate than the prediction of lower peaks, as if the neural network tends to learn more about the max critical vibration value, which is favorable for this project.

Results of Architecture II

The results of this architecture, shown in Table 5.3, came out to be the least accurate in vibration prediction. While it managed to predict much of the vibration, its performance was weak at the peaks (either low or high) compared to the other architectures, as shown in Figures 5.3, and 5.4. However, the lower peaks were better at some positions on the curve of this architecture, compared to the other architectures. A potential reason for the poor performance of this architecture is due to using the average of values from the LSTM second layer output, the other two architectures can weight the values from the LSTM second layer output for more accuracy.

Results of Architecture III

This LSTM RNN was one layer deeper and also had 20 seconds memory from the past, which was not available for the other two LSTM RNNs used. Although this architecture was the most computationally expensive and thus had the most chance for deeper learning, the results of this architecture were not as good as expected, as shown in Figures 5.3, and 5.4. The results of this architecture in Table 5.3 show that the prediction accuracy for this architecture was less than the more simple Architecture I.

The overall error in Table 5.3 for the prediction at 20 future seconds was relatively high. At some locations in the plotted curve for this architecture predicting vibrations

at 20 seconds in the future, the calculated curve became much higher than the actual vibration curve. This strange behavior is unique as it can be seen that the calculated vibration would rarely exceed the actual vibration for all the curves plotted for all the architectures at all scenarios, and it would be for relatively small value if occurred. The performance of this architecture (the mean absolute error) was slightly better than the other architectures when predicting for 1 second in the future; however, it performed worse in the other time scales. While a more complex architecture should have more potential to "learn" and perform better predictions, it appears that the challenge of training this architecture negated most of these benefits. This is reinforced by Figure 5.2, which shows that the error of this architecture did not decrease smoothly while trained to predict for 20 seconds in the future. To investigate if this network could potentially gain further improvement if trained for more epochs, it was retrained for 1150 instead of 575 epochs. However, this failed to result in any significant improvement in predictive ability. This could potentially be a result of other issues in the training process due to a more complex search space.

5.2.6 Memory Cell Ant Colony Optimization (MC-ACO) Results

For the fixed networks, Architecture I gave the most promising results, so it was chosen as the initial candidate for MC-ACO. The MC-ACO code was run for 1000 iterations using 200 ants. The networks were allowed to train for 575 epochs to learn and for the error curve to flatten. The minimum value for the pheromones was 1 and the maximum was 20. The population size was equal to number number of iterations in the ACO process, *i.e.*, the population size was also 1000. Each run took approximately 4 days.

For a more rigorous examination, the 57 flights in the vibration dataset were divided into 10 subsamples. The subsamples were used to cross validate the results by examining combinations utilizing nine of the subsamples as the training data set and the tenth as the testing set.

These subsamples were used to train the NOE, NARX, NBH, Architecture I and the ACO Architecture I. Figures 5.10 shows predictions for the different models over a selection of test flights, and Figure 5.11 shows predictions of a single uncompressed (higher resolution) test flight. Table 5.6 compares these models to Architecture I (LSTM) and the ACO optimized Architecture I (ACO). Figure 5.9 shows box-plot for the results

shown in Table 5.6.

NOE, NARX, and NBJ Results

Somewhat expectedly, the NOE model performed the worst with with a mean error of 15.73% ($\sigma = 0.0941$). The NBJ model performed better than the NOE model with a mean error of 15.05% ($\sigma = 0.1338$); however, the NARX model performed better than the previous two models with a mean error of 12.06 % ($\sigma = 0.0663$). This is interesting in that the NBJ model had access to actual future vibration values, unlike NOE, NARX and the LSTM models; and could be expected to perform better utilizing this information. The discrepancy in performance is likely due to the high nonlinearity in the input and target parameters, along with the difficulty of training RNNs on long time series data.

Architecture I Revisited

The Architecture I RNN was trained utilizing the ten subsamples to validate the results. The mean error for each of the ten subsamples (using the other two as training data) was 5.61% ($\sigma = 0.0245$).

Network Regularization

Regularization [164] was implemented on the investigated LSTM, NOE, NARX, and NBJ networks to validate ACO. The used connection-dropout percent is 30%. This percent was chosen because the number connection subject to regularization is not very large. The results were as follows: *a) LSTM Regularization:* the obtained mean error is 7.62% ($\sigma = 0.0183$), *b) NOE Regularization:* the obtained mean error is 8.70% ($\sigma = 0.0021$), *c) NARX Regularization:* the obtained mean error is 9.40% ($\sigma = 0.0013$), and *d) NBJ Regularization:* the obtained mean error is 9.43% ($\sigma = 0.0020$).

Ant Colony Optimized Architecture I

When ACO optimization was used to find the optimal connections to keep in the Architecture I RNN, the best version of Architecture I evolved with ACO showed an improvement of 1.34% for predictions 10 seconds in the future, reducing prediction error from 5.61% to 4.27% compared to the architecture's performance before the ACO. Figure 5.5a provides an example of the improvement in predictions on a single test flight, before and after the ACO optimization.

The topology of the best networks' cells are shown in Figures 5.6 and 5.7. The ACO generated mesh used to generate this topology is shown in the matrices in Equations 5.3 and 5.4. It is worth stressing that this topology is not the complete LSTM RNN used in the utilized Architecture I, but rather applies to the individual gates in each cell. Equation 5.3 is used for any fully connected process and Equation 5.4 is used for any data-reduction process (discussed in detail in Chapter 4.1.2). Figures 5.6a, 5.6b, and 5.6c show the ACO optimized $mesh\_1$, which were used within the "M1" LSTM cells (see Figure 4.4) in the top ten evolved LSTM RNNs. Figure 5.8 provides an example of how the M1 cells are updated with these connections. While the connections reduction did not show any inputs being fully eliminated, which was sought as one of the goals of the study, a significant number of connections were removed.

The evolved networks retained all the elements of $mesh\_2$, represented by Equation 5.4, for use in the "M2" LSTM cells (see Figure 4.5). Figure 5.7 is used to show this part of the evolved mesh. For clarity, Figure 5.8b shows the differences between the M1 cells before and after ACO optimization. Figure 5.8a is simply a LSTM cell "M1" that have its gates' meshes (shown in Figure 5.8b, Up) substituted with the ACO meshes (shown in Figure 5.8b, Down). "M2" did not change from its original topology as shown in Figure 4.5 since all the elements in $mesh\_2$ after the optimization remained ones (Equation 5.4).

The colored nodes in Figure 5.6a are the input nodes (first line of nodes) at the Main, Input, Forget, and Output gates at the "M1" cells (Figure 4.4). The diamond nodes in Figure 5.6a are the hidden layer nodes (second line of nodes) at the Main, Input, Forget, and Output gates at the "M1" cells (Figure 4.4). The diamond nodes are also the input nodes at the Main, Input, Forget, and Output gates at the"M2" cells (Figure 4.5). The last single node in Figure 5.7 is the output of the gates in the "M2" cells.

Returning to an initial question of how the number of the connections in the network affects the soundness of the results, Table 5.5 shows the top 30 evolved networks with respect to the fitnesses they provide. The table also shows the total number of connections in both $mesh\_1$ (first set of connections in the generated mesh) and $mesh\_2$ (second set of connections in the generated mesh), and the total number weights (connections) in the networks. Comparing these values to the total number of weights in a fully connected Architecture I type network, as shown in Table 4.2, it is found that total number of weights were reduced by 42% to 45% in the top 30 networks.

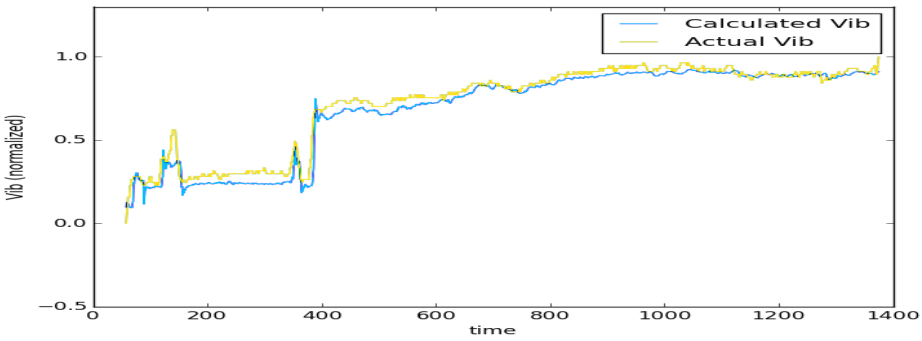
The complete experiments and results were published in this study [39].

$$mesh\_1 = \begin{array}{r|cccccccccccccccc|} & h1 & h2 & h3 & h4 & h5 & h6 & h7 & h8 & h9 & h10 & h11 & h12 & h13 & h14 & h15 & h16 \\ i1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ i2 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ i3 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ i4 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ i5 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ i6 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ i7 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ i8 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\ i9 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ i10 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ i11 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ i12 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ i13 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\ i14 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ i15 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ bias & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \end{array} \tag{5.3}$$

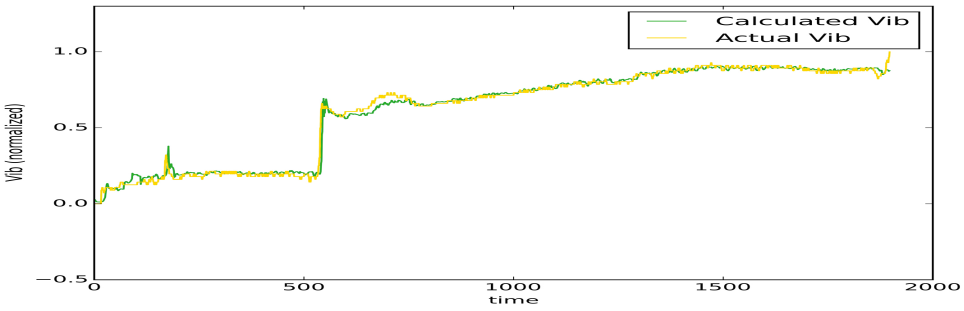
$$mesh\_2 = \begin{array}{|cccccccccccccccc|} h1 & h2 & h3 & h4 & h5 & h6 & h7 & h8 & h9 & h10 & h11 & h12 & h13 & h14 & h15 & h16 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{array} \tag{5.4}$$

Table 5.5: ACO Top Thirty Evolved Networks

No.	Fitness	Number of M1 Connections	Number of M2 Connections	Total Number of Connections	No.	Fitness	Number of M1 Connections	Number of M2 Connections	Total Number of Connections
1	0.034888	137	16	11650	16	0.036795	140	16	11890
2	0.034917	136	16	11570	17	0.036820	145	16	12290
3	0.035851	141	16	11970	18	0.036901	140	16	11890
4	0.036063	146	16	12370	19	0.036932	131	16	11170
5	0.036067	143	16	12130	20	0.036953	142	16	12050
6	0.036337	140	16	11890	21	0.037001	141	16	11970
7	0.036535	136	16	11570	22	0.037040	145	16	12290
8	0.036582	140	16	11890	23	0.037041	147	16	12450
9	0.036588	133	16	11330	24	0.037082	133	16	11330
10	0.036647	134	16	11410	25	0.037106	142	16	12050
11	0.036715	135	16	11490	26	0.037114	135	16	11490
12	0.036727	143	16	12130	27	0.037134	137	16	11650
13	0.036730	147	16	12450	28	0.037142	138	16	11730
14	0.036787	143	16	12130	29	0.037145	144	16	12210
15	0.036788	137	16	11650	30	0.037161	139	16	11810

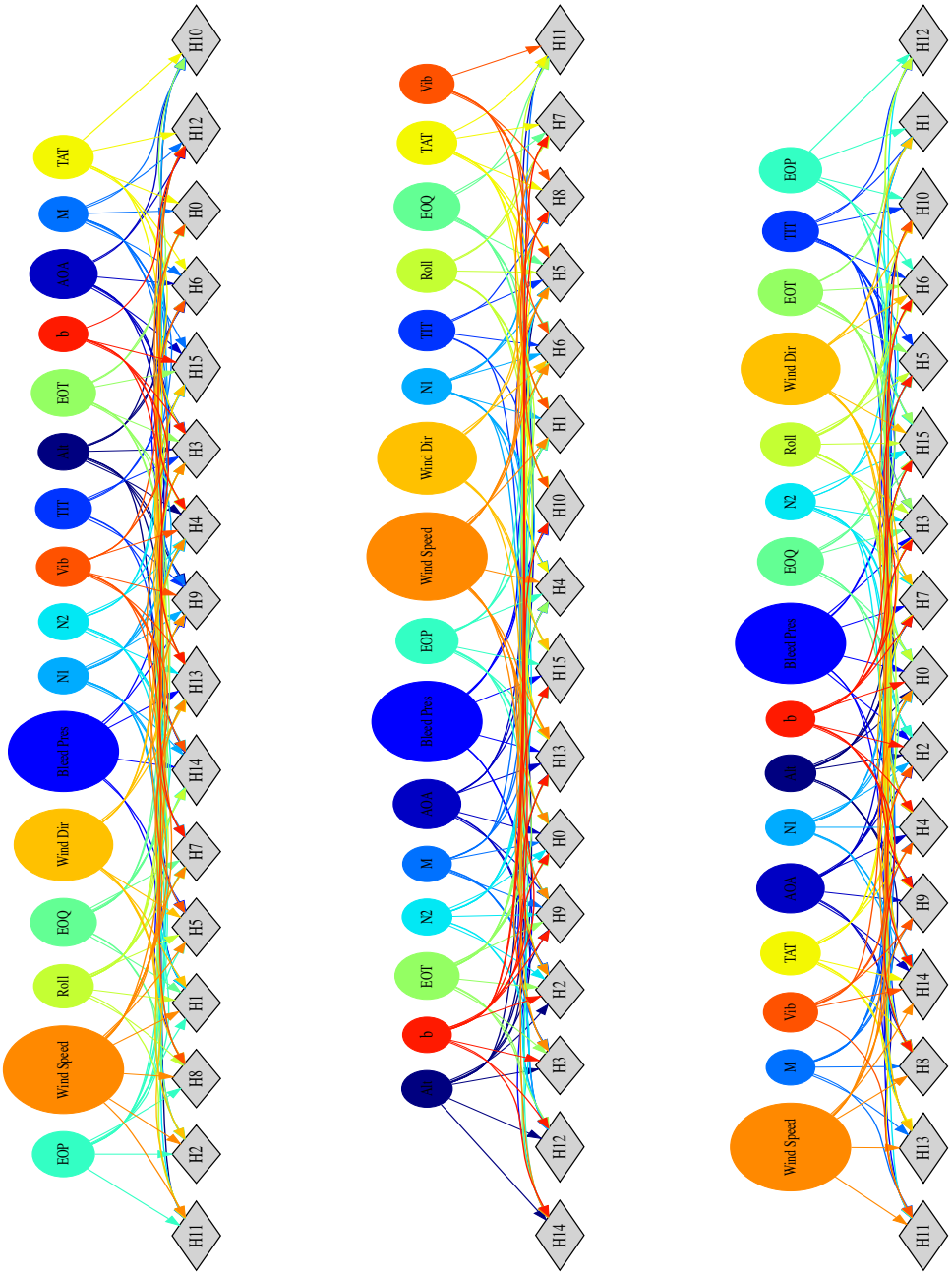


(a) *Unoptimized: single test flights*



(b) *Optimized: single test flights*

Figure 5.5: *Plotted results for predicting ten seconds in the future.*



(a) *ACO Architecture I First Best Fitness Mesh: 155 connections.*

(b) *ACO Architecture I Second Best Fitness Mesh: 152 connections.*

(c) *ACO Architecture I Third Best Fitness Mesh: 160 connections.*

Figure 5.6: *ACO Architecture I Best Fitness Topologies' Meshes (Equation 5.3) for at "M1" (Figure 4.4) LSTM cells: 1000 Iterations, and 200 Ants.*

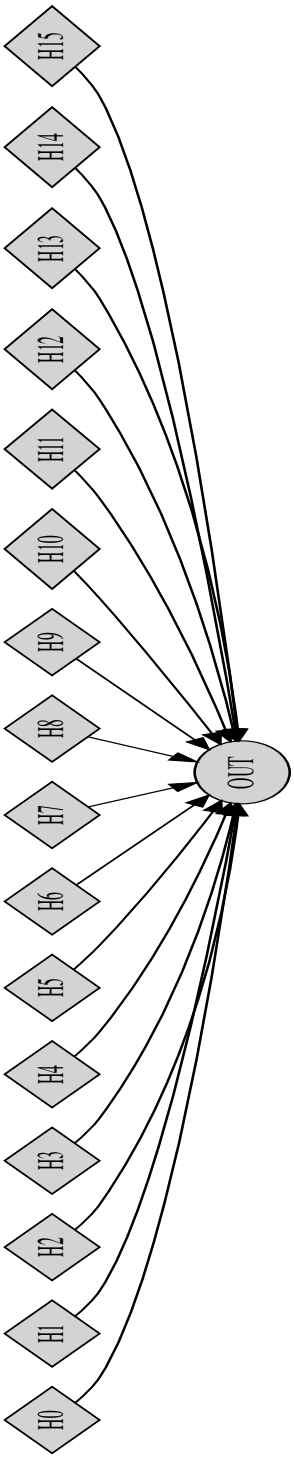
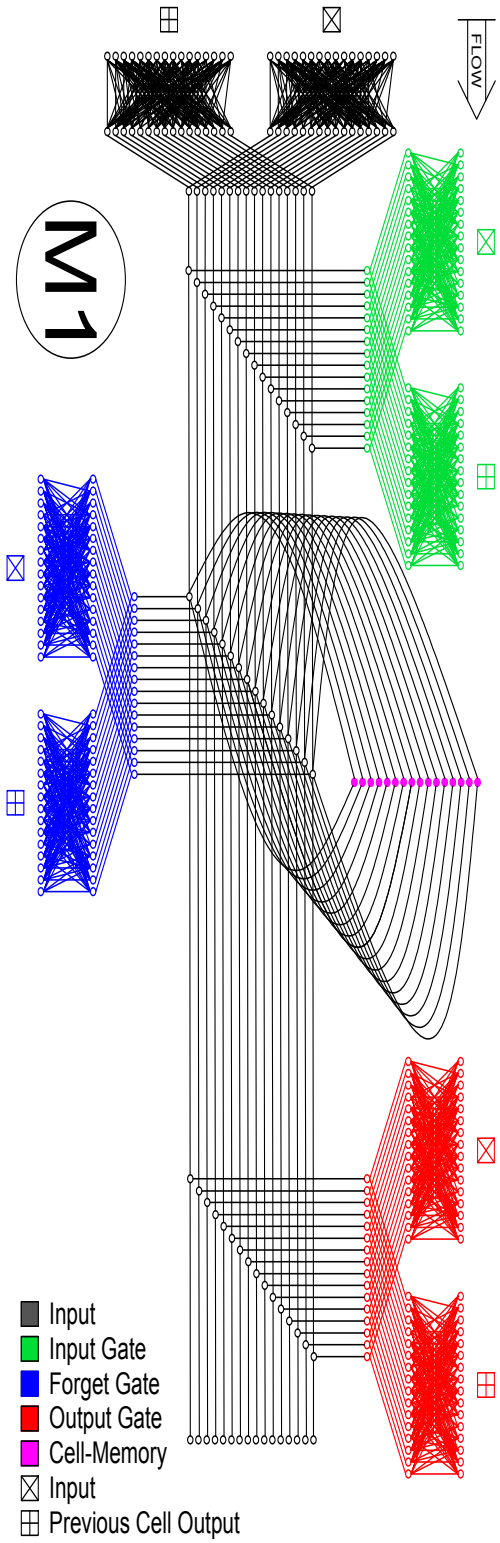
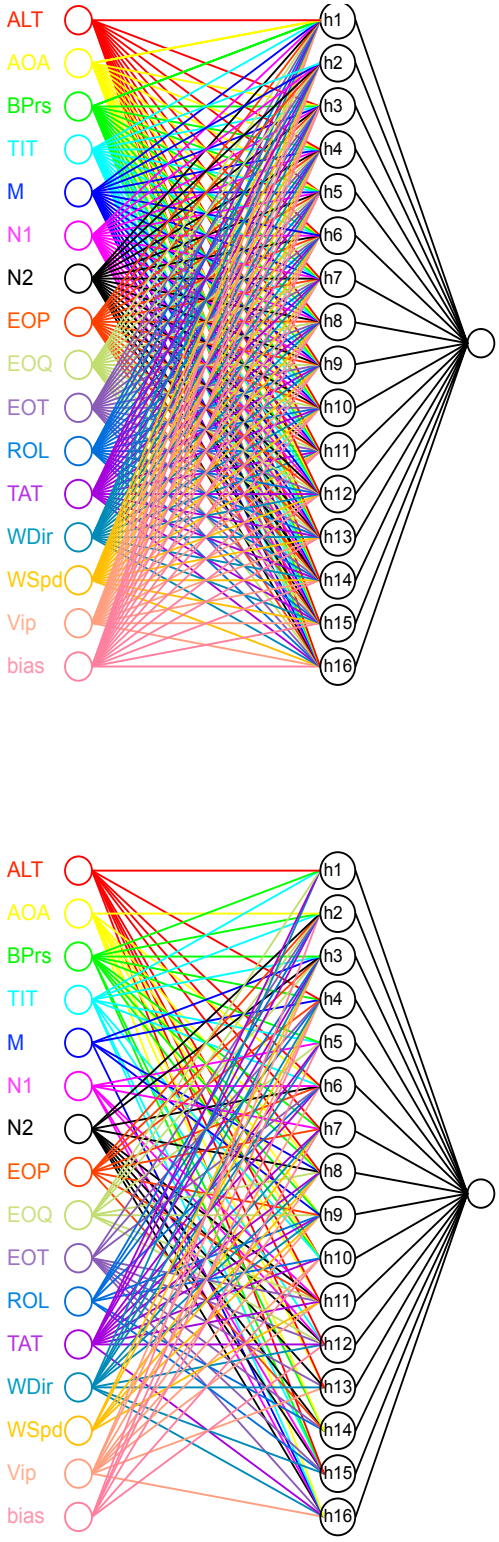


Figure 5.7: *ACO Architecture I Best Fitness Topology's mesh (Equation 5.4) at "M2" (Figure 4.5) LSTM cells: 1000 Iterations, and 200 Ants.*



(a) *ACO Architecture I First Best Fitness Network: LSRM M1 cell.*



(b) *Meshes before (up) and after (down) optimization.*

Figure 5.8: *An example of the M1 cell before and after optimization.*

Table 5.6: 10-Fold Cross Validation Results

	Prediction Errors (MAE)								
	LSTM	**LSTM_Reg**	**NOE**	**NARX**	**NBJ**	**LSTM_ACO**	**NOE_Reg**	**NARX_Reg**	**NBJ_Reg**
Fold 1	8.98%	5.68%	12.45%	10.1%	9.28%	8.11%	9.14%	8.63%	9.24%
Fold 2	3.68%	5.84%	16.37%	16.44%	15.88%	3.39%	9.63%	8.58%	9.52%
Fold 3	7.23%	6.35%	14.19%	8.07%	7.81%	5.65%	9.25%	8.30%	9.41%
Fold 4	3.71%	7.95%	8.56%	5.18%	6.21%	4.15%	9.42%	8.70%	9.38%
Fold 5	10.76%	6.89%	15.13%	15.67%	20.10%	5.87%	9.56%	8.58%	9.41%
Fold 6	5.93%	9.40%	13.36%	7.56%	7.92%	4.05%	9.78%	8.73%	9.47%
Fold 7	3.46%	9.34%	18.03%	10.44%	13.02%	2.69%	9.60%	8.65%	9.19%
Fold 8	3.58%	5.29%	42.14%	29.27%	53.16%	3.03%	9.32%	9.01%	9.53%
Fold 9	4.15%	11.11%	10.47%	7.92%	10.57%	3.35%	9.40%	8.75%	9.60%
Fold 10	4.62%	8.33%	6.61%	9.97%	6.55%	2.43%	9.21%	9.06%	9.28%
Mean	**0.0561**	**0.0762**	**0.1573**	**0.1206**	**0.1505**	**0.0427**	**0.0870**	**0.0940**	**0.0943**
Std. Dev.	**0.0245**	**0.0183**	**0.0941**	**0.0663**	**0.1338**	**0.0168**	**0.0021**	**0.0013**	**0.0020**

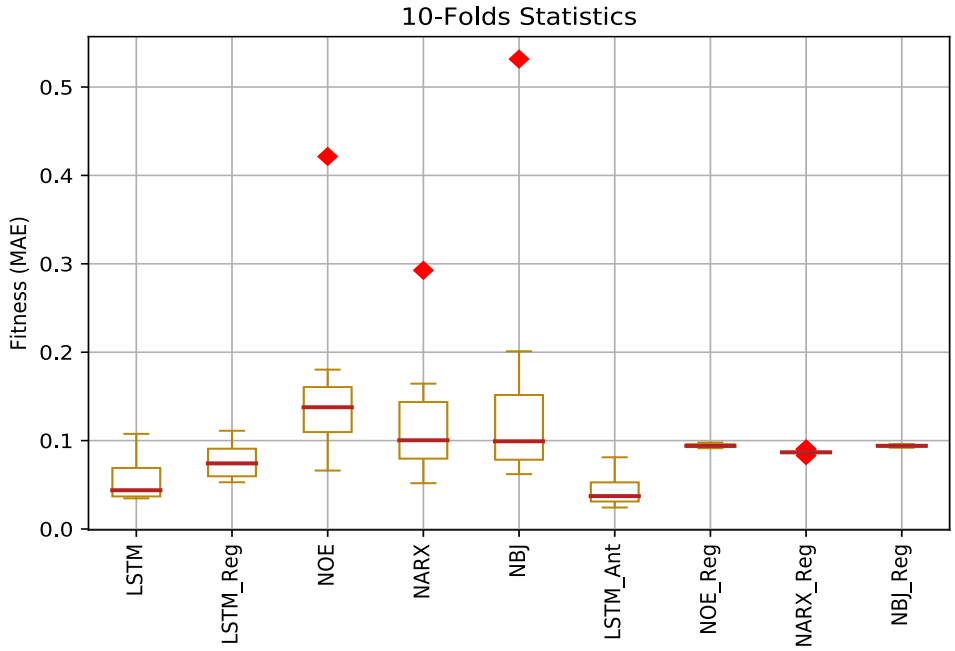
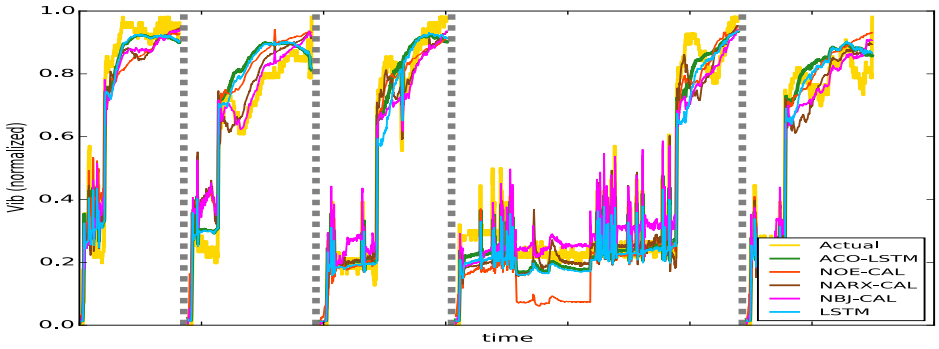
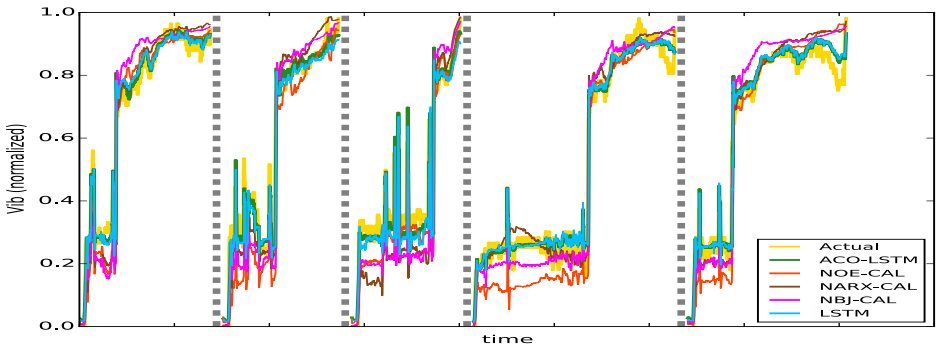


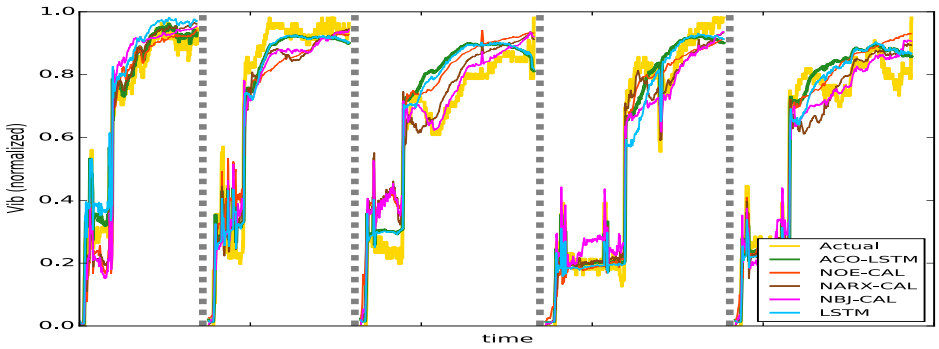
Figure 5.9: *Box-plot: 10-Fold Cross Validation Results*



(a) *Subsample 1*

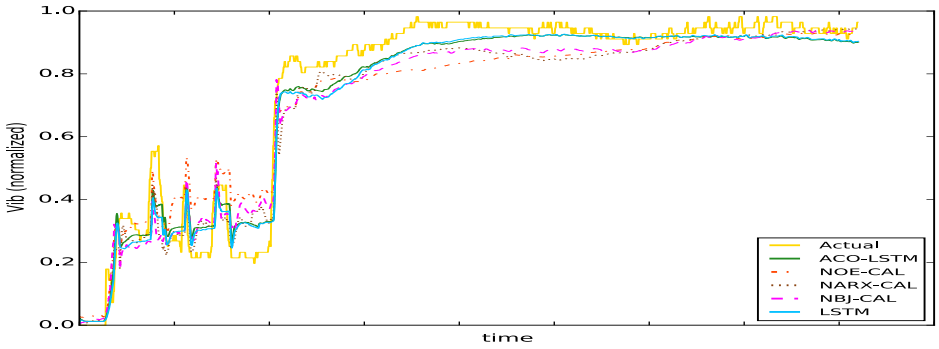


(b) *Subsample 2*

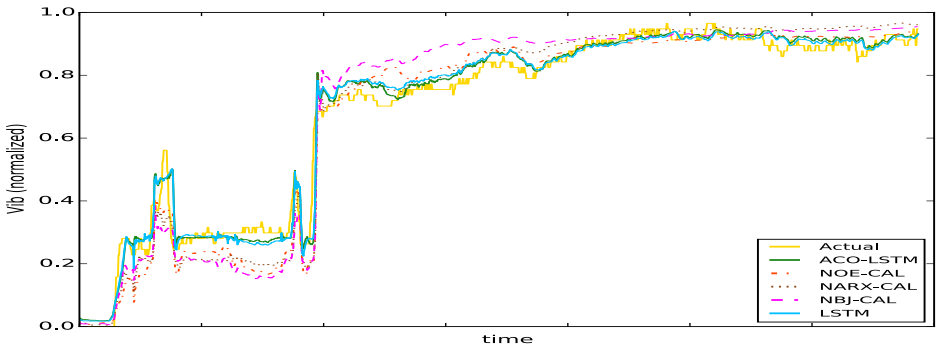


(c) *Subsample 3*

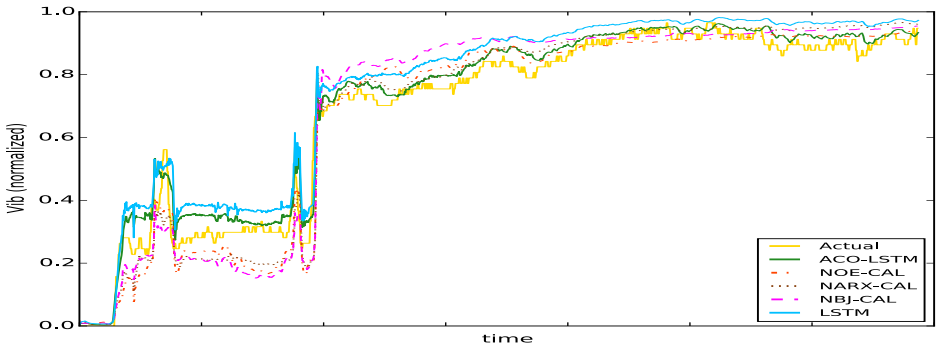
Figure 5.10: *Results for the K-fold cross validation subsamples predicting vibration ten seconds in the future for a selection of 4 flights.*



(a) *Subsample 1*



(b) *Subsample 2*



(c) *Subsample 3*

Figure 5.11: *Results for the K-fold cross validation subsamples predicting vibration ten seconds in the future for an individual flight.*

5.3 ANTS Results

ANTS was compared to both NEAT and EXAMM, as well as traditional layered RNN architectures. All ANTS and EXAMM experiments generated 2000 total RNNs, training each for 10 epochs. NEAT, on the other hand, was allowed to generate 420,000 RNNs. If we assume that a forward pass (forward propagation) and a backward pass (backprop calculation) are approximately the same computationally, this generously gave NEAT approximately 10 times the amount of compute time (as 2000 RNNs trained for 10 epochs would equivocate to 20,000 forward and 20,000 backward passes). The RNNs with non-evolvable (fixed) architectures were allowed to train for 70 epochs. Every experiment was repeated 10 times to compute means and standard deviations in order to ensure a proper statistical comparison.

ANTS used a colony superstructure with 12 input nodes, 3 hidden layers, each with 12 hidden nodes, and a single output node. Recurrent synapses could span 1, 2 or 3 steps in time. The resulting connectivity superstructure consisted of 49 nodes, 924 edges, and 3626 recurrent edges. While this may seem modest compared to modern convolutional architectures, which may consist of millions of connections, it is important to note that the RNNs generated from this superstructure are unrolled over 7200 time steps (according to the time series length of the training and testing data samples) when trained locally via backpropagation through time (BPTT). That is, algorithms such as ANTS must handle (fully-unrolled) networks of up to 3,528,000 nodes, 6,652,800 edges, and 26,107,200 recurrent edges with errors from the final output (predictor) potentially back-propagated over up to 28,000 synaptic connections.

The dataset utilized in this study was an open access time series dataset taken from a coal fired powerplant. The data was introduced in previous neuro-evolution studies for time series data prediction [38, 109]. It consists of 12 possible parameters, recorded for 10 days with each parameter recorded at each minute. These 12 parameters were used to predict the flame intensity parameter (the response variable, in regression parlance). Results were generated by training RNNs on 5 days worth of data taken from one of the coal burners from this dataset. Fitness values (mean absolute error) were calculated on the other 5 days (test set). 1,600 experiments were conducted in order to include all combinations of the ANTS options/variations (described below). Each experiment was repeated 10 times to obtain robust results.

These ANTS experiments generated, trained, and evaluated 32 million RNNs. Overall, the experiments took approximately 30 days to complete. Given the unstructured nature of the RNNs evolved in this work, utilizing CPUs has been found to be more efficient than GPUs as there are no wide, fully connected layers which would benefit from parallelized matrix algebra on a GPU. Further, it allows the use of large scale high performance computing clusters which typically have many more CPUs than GPUs available.

5.3.1 Backpropagation Hyperparameters

All ANNs were trained with backprop and stochastic gradient descent (SGD) using the same hyperparameters. SGD was run with a learning rate $\eta = 0.001$ and used Nesterov momentum ($mu = 0.9$) to smooth out the local gradient descent. No dropout regularization was used since it has been shown in other work to reduce performance when training RNNs for time series prediction as shown in Section 5.2 and the published article for this work [39]. To prevent exploding gradients, gradients were re-scaled using gradient clipping (as prescribed by Pascanu *et al.* [114]) when the norm of the gradient was above a threshold of 1.0. To improve performance in the case of vanishing gradients, gradient boosting (the opposite of clipping) was used when the norm of the gradient was below a threshold of 0.05. The forget gate bias of the LSTM cells had 1.0 added to it as this has been shown to yield significant improvements in training time by Jozefowicz *et al.* [75]. Weights for RNN in all other cases were initialized as described in the section describing the communal weight sharing 4.2.1 scheme for ANTS and or by EXAMM's Lamarckian weight inheritance [109].

5.3.2 ANTS Options and Hyper-parameters

The influence/effect of individual ANTS hyper-parameters was carefully investigated in this study. A pheromone decay rate of $\alpha = 0.05$ and a pheromone evaporation rate of $\beta = 0.1$ were chosen as they were shown to be effective in preliminary tests and is within the recommended standard range [131]. The other considered ANTS parameters were:

1. Number of ants : {20, 40, 80, 160}.

2. Regularization update parameter: {0.25, 0.65, 0.90}.
3. Initializing RNN using communally shared weights with constant Φ values of ({0.3, 0.6, 0.9}), using Φ as calculated by a function of fitness, and basic randomized weight initialization.

The application of the examined heuristics that appear in the figures and tables that follow are labeled as follows:

1. Function Φ : $\Phi()$
2. Constant Φ: $\Phi_{value\ of\ \Phi}$
3. L1 Pheromone regularization: $L1_{value\ of\ \gamma}$ (Equation 4.11)
4. L2 Pheromone regularization: $L2_{value\ of\ \gamma}$ (Equation 4.12)
5. Standard Ant Species: Without Bias (Std) and With Bias ($StdBias$)
6. Multi Species Ants:
   - Explorer Ants: Exp
   - Explorer Ants and Forward Social Ants: $ExpFwd$
   - Explorer Ants and Backward Social Ants: $ExpBwd$
   - Explorer Ants, Forward and Backward Social Ants: $ExpFwdBwd$
7. Layer Jumping: AJ
8. No Layer Jumping: OJ

5.3.3 ANTS Preliminary Results

ANTS heuristics were tested first individually by applying one heuristic at time while not utilizing any others. Figure 5.12 presents the performance of ANTS when each each heuristic is applied separately. Furthermore, the results compare the performance of the solo application of ANTS heuristics to the state-of-the-art EXAMM, NEAT, and traditional fixed standard RNNs. While ANTS in this case (augmented only by individual heuristics) did not outperform EXAMM except for some outliers, both EXAMM

and ANTS showed dramatically better performance than NEAT, even though NEAT was given a significant amount of extra compute time. ANTS, EXAMM and NEAT also significantly outperformed traditional RNNs. Some of the gain over NEAT is most likely due to the use of backpropagation by EXAMM and ANTS since NEAT uses fairly simple and non-gradient based recombination operations to adjust weights.

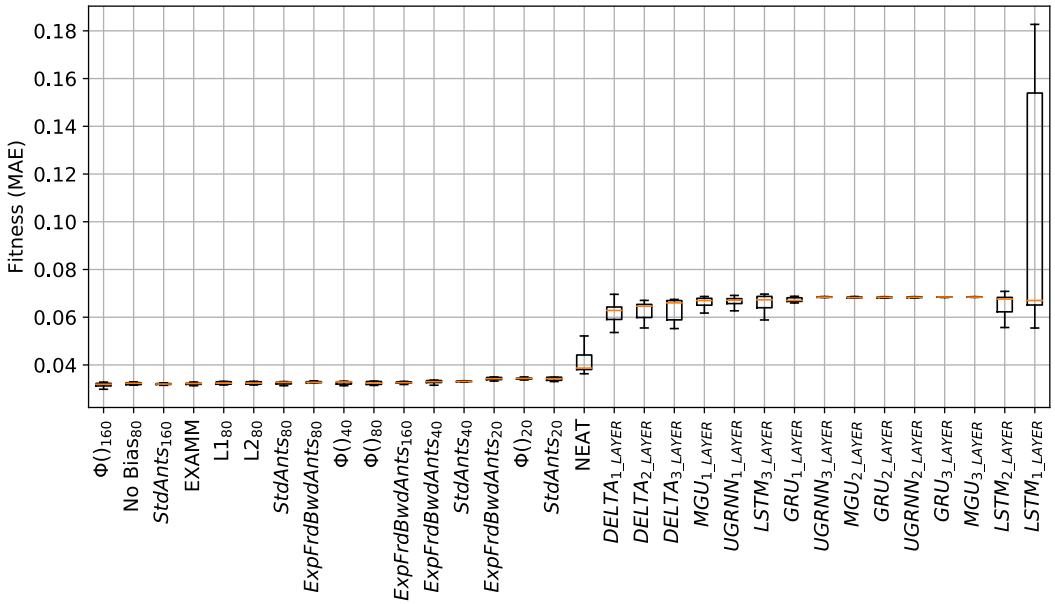
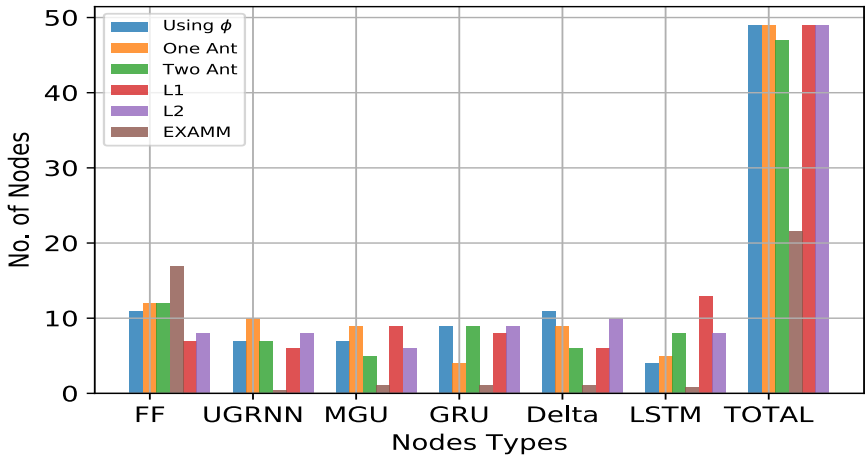


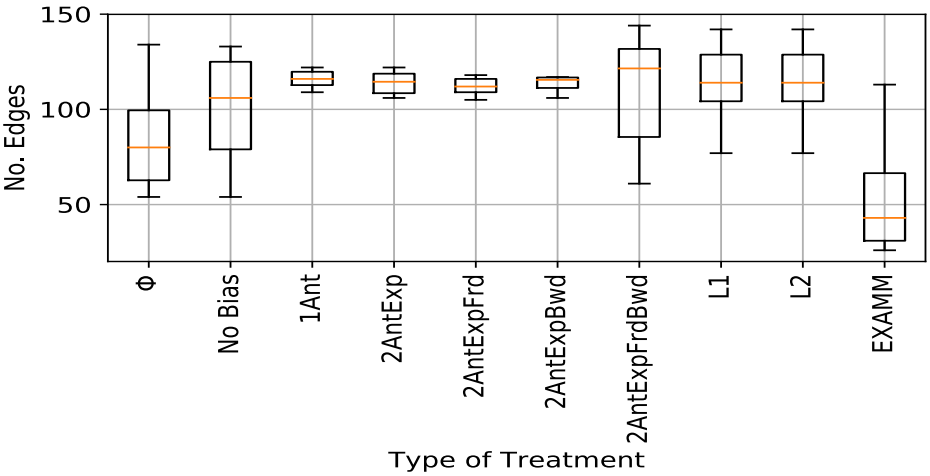
Figure 5.12: *Performance of NEAT, EXAMM, & individually applied ANTS heuristics against fixed memory cell RNNs.*

5.3.4 Generated Structures

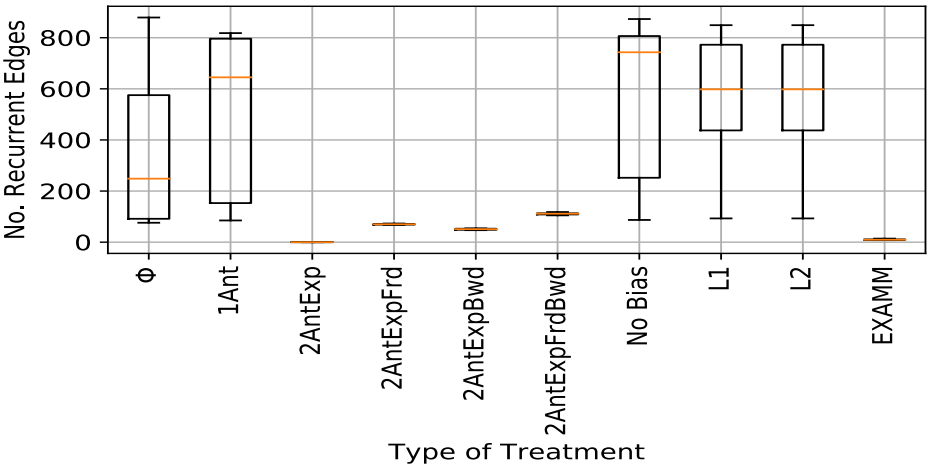
Comparing the generated best performing structures of EXAMM and ANTS shows a disparity in the sizes of those structures. While EXAMM maintained its structures relatively small, ANTS tended to generate larger structures with more nodes, edges, and recurrent edges as shown in Figures 5.13. This opens door to questions about the local minimas which EXAMM might fall into or that the optimization iterations of ANTS and EXAMM need to be increased to converge to similar structures size-wise. Figures 5.13 also show that the two-ant-species heuristic of ANTS which targeted the size of the generated structure was very effective in reducing the recurrent connections, especially when compared to EXAMM as a reference.



(a) *Structures Number of Nodes*



(b) *Structures Number of Edges*



(c) *Structures Number of Recurrent Edges*

Figure 5.13: *ANTS Optimized Structural Elements*

5.3.5 Performance of Combined Heuristics

	Top 10			Top 25			Top 100			Top 250		
	Mean	Median	Best	Mean	Median	Best	Mean	Median	Best	Mean	Median	Best
Φ()	3(0)	4(0)	3(0)	9(0)	7(0)	9(0)	26(0)	23(0)	31(8)	58(0)	54(0)	49(8)
ConstΦ	7(0)	6(0)	7(0)	14(0)	14(0)	12(0)	60(0)	63(0)	54(8)	147(0)	149(0)	155(16)
NoΦ	0(0)	0(0)	0(0)	2(0)	4(0)	4(0)	14(0)	14(0)	15(0)	45(0)	47(0)	46(0)
L1	2(0)	4(0)	0(0)	9(0)	8(0)	3(3)	42(0)	34(0)	30(4)	96(0)	96(0)	91(4)
L2	5(0)	5(0)	6(0)	13(0)	12(0)	16(1)	40(0)	45(0)	38(3)	100(0)	98(0)	95(12)
StdAnts	0	0	0	1	0	0	3	0	0	20	19	0
StdBiasAnts	0	0	0	0	0	0	3	1	0	23	16	0
ExpAnts	0	0	10	0	0	25	1	0	100	10	6	250
ExpFdAnts	6	7	0	14	15	0	45	49	0	98	103	0
ExpBwAnts	0	0	0	0	0	0	0	0	0	0	0	0
ExpFBAnts	4	3	0	10	10	0	48	50	0	99	106	0
No Jump	0	0	5	0	0	13	0	0	52	0	0	128
Layer Jump	10	10	5	25	25	12	100	100	48	250	250	122
20 Ants	0	0	2	0	0	6	0	0	24	0	0	65
40 Ants	2	0	3	5	1	7	14	15	23	50	57	63
80 Ants	4	3	2	8	11	6	44	45	26	82	80	60
160 Ants	4	7	3	12	13	6	42	40	27	118	113	62

Table 5.7: Heuristic Ranking Statistics

The combined application of multiple different heuristics, as illustrated in Figure 5.14, yielded ANTS results that outperformed *all baselines*, including the fixed RNNs, NEAT, *as well as* EXAMM. Table 5.7 provides statistics ranking each of the heuristics based on how many times the experiments that utilized them appeared in the top 10, 25, 100, and 250 best results as determined by the mean, median, and the best performance of the RNN generated in the experiment's 10 repeats. Values in parentheses are the number of times an experiment that only utilized that single heuristic appeared in that top ranking. The utilization of multiple heuristics dominated the top results, with individually-applied heuristics not appearing in the top 10, and only 4 times in the top 25 (only as best results).

Communal weight sharing proved to be very important, yielding strong performance, with all of the top 10 utilizing either functional or constant Φ parameters. Furthermore, it also occurred 2 (mean), 4 (median) and 4 (best) times in the top 10, and 14 (mean), 14 (median), and 15 (best) times in the top 25. Additionally, all of the best performing RNNs used layer-jumping ants, which tend favor more sparse connectivity patterns. Most of the best results used pheromone weight-regularization, with L2 regularization appearing at a nearly 50% rate in the top 10, 25 and 100 results. The regularization factor was also high, at 65% or 90%, for most of the 25 best experiments that used it.

All of the top 250 best results utilized the multiple ant species heuristic, which strongly

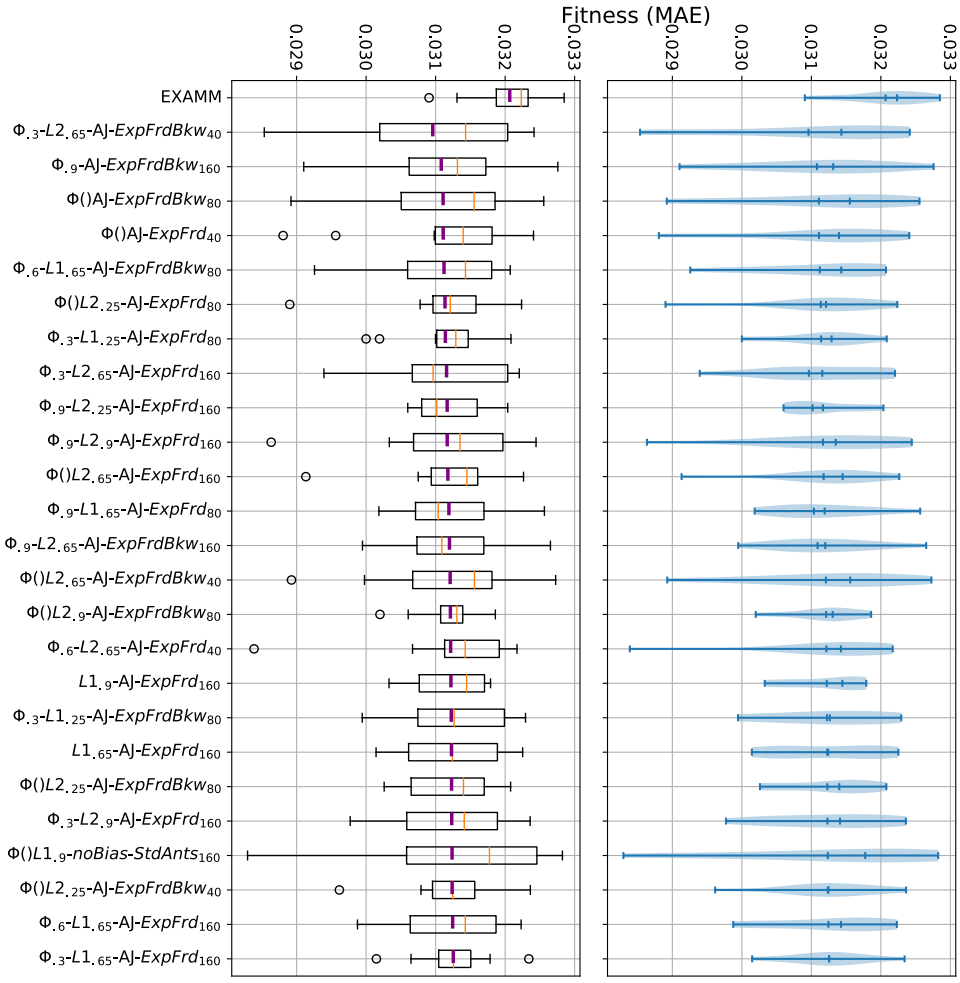


Figure 5.14: *Performance of EXAMM and the top 25 ANTS experiments*

supports the use of specialized ants. The number of ants varied between 20 and 160 for all the top 25 results in the mean and median case, with a larger number of ants tending to perform better. However, the case of 20 ants did occasionally appear in the best cases, even sometimes in the top 10. Moreover, these networks tended to be rather sparse but very well performing. This may suggest that the experiments that utilized more ants had an easier time finding the most important structures, but also potentially had extraneous connections which were not needed. In contrast, the experiments with less ants had less of a chance of finding these important structures due to lower (overall) connectivity. This suggests that further optimizations could be designed to better guide ANTS towards the discovery of more efficient networks.

Perhaps one the most interesting items to observe is the performance distribution when multiple ant agent roles was used in ANTS. The entirety of the best found RNNs, up to the top 250 were from explorer ants only, so these generated RNNs only had recurrent connectivity in terms of whatever the various memory cells offered. However, for the mean and median performance of the experiments, nearly all the top 25, 100, and 250 consisted of explorer and forward recurrent roles or explorer, forward, and backward recurrent ant specializations – with only a very few of the only explorer ant only configurations showing up in the top 100 and 250. First, this suggests that backward recurrent connections (which are most commonly utilized in RNNs) were less effective than forward recurrent connections. Second, it also appears that adding these recurrent connections tended to make the RNNs perform significantly better on the average and median cases, while the RNNs generated with only explorer ants had the ability to occasionally find RNNs that generalized quite well. These results certainly suggest further study in order to better understand the effect of combining recurrent connections and memory cells. In addition, alternative strategies can be developed that retain the stability of adding recurrent connections while still efficiently finding well-generalizing RNNs. The complete experiments results are published in this study [43].

5.4 CANTS Results

CANTS' results offer an insight about how effective this method compared to its discrete search space sister, ANTS, and the other NAS methods represented by EXAMM. The experiments compare CANTS to the state-of-the-art ANTS and EXAMM algorithms on three real world datasets related to power systems. All three methods were used to perform time series data prediction for different parameters, which have been used as benchmarks in prior work. Main flame intensity was used as the prediction parameter from the coal plant's burner, net plant heat rate was used from the coal plant's boiler, and average power output was used from the wind turbines. Experiments were also performed to investigate the effect of CANTS hyper-parameters: the number of cants and cant sensing radii, ϵ.

5.4.1 Number of Cant Agents

An experiment was conducted to determine the effect that the number of cant agents has on the performance of CANTS. The experiment focusing on the net plant heat rate feature from the coal-fired power plant dataset. The number of ants evaluated were 10, 30, 60, 100, 150, and 210. The results, illustrated in Figure 5.15, show that, as the number of cants are increased, the performance increases until 150 cants are used and then a decline is observed. This shows that the number of cant agents is an important parameter and requires tuning, potentially exhibiting "sweet spots" that, if uncovered, provide strong results.

5.4.2 Cant Agent Sensing Radius

The effect that the sensing radii (range) of the cant agents had on algorithm performance was investigated next. Several cant sensing radii were evaluated to observe the effect of having cants with different sensing ranges. Figures 5.16 shows that a sensing radius of 0.5 showed better performance compared to the 0.1, 0.2, 0.3, 0.4, and 0.6 sensing radii values we tested. In addition to this, using a randomly generated sensing radius per cant agent was also evaluated. For these, ϵ was randomly initialized (uniformly) via $\sim U(0.01, 0.98)$. Ultimately, we discovered that the sensing radius of 0.5 still provided the best results.

5.4.3 Algorithm Benchmark Comparisons

To compare the three different NAS strategies, each experiment was repeated 10 times (trials) for statistical comparison and all algorithms were set to generate 2000 RNNs per trial. For CANTS, the sensing radii of the cant agents and exploration instinct values were generated uniformly via $\sim U(0.01, 0.98)$ when the cants were created, initial pheromone values were 1 and the maximum was kept at 10 with a pheromone decay rate set to 0.05. For the DBSCAN module, clustering distance was 0.05 with a minimum point value of 2 – runs with these settings were done using 30 and 150 ants. CANTS and ANTS used a population of size 20 while EXAMM used 4 islands, each with a population of 10. ANTS, CANTS, and EXAMM all had a maximum recurrent depth of 5 and the predictions were made over a forecasting horizon of 1. The generated

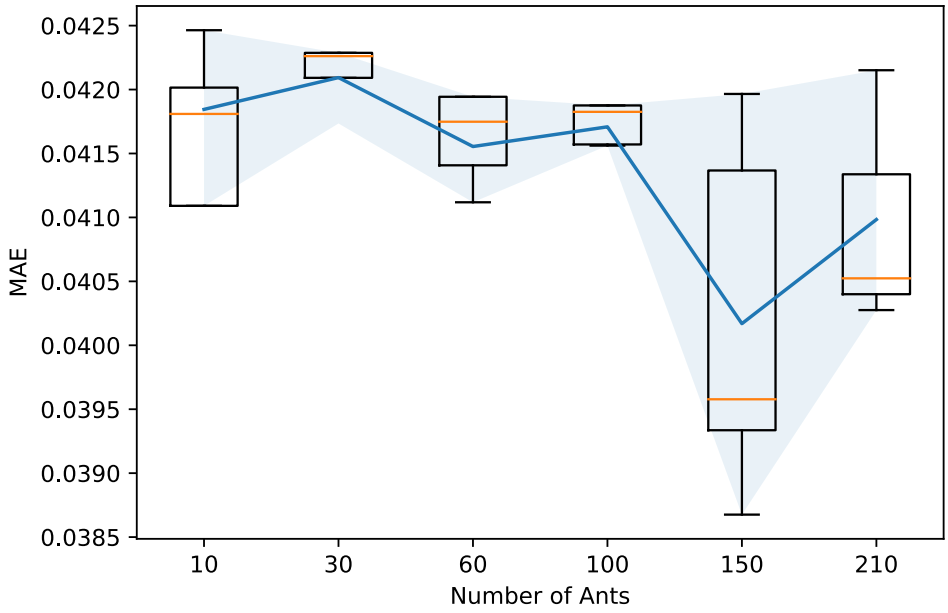


Figure 5.15: *CANTS with a varying number of agents.*

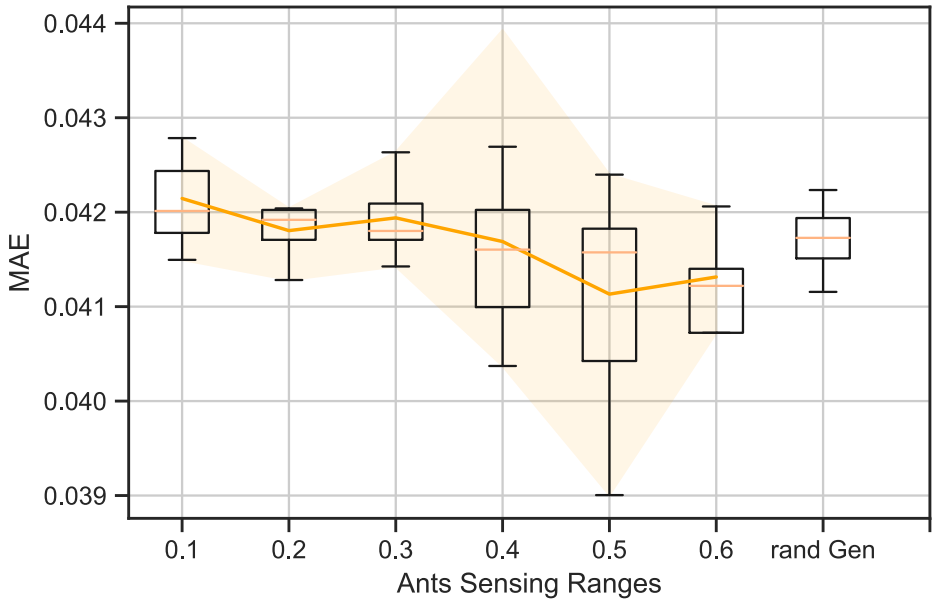


Figure 5.16: *CANTS with different sensing radii.*

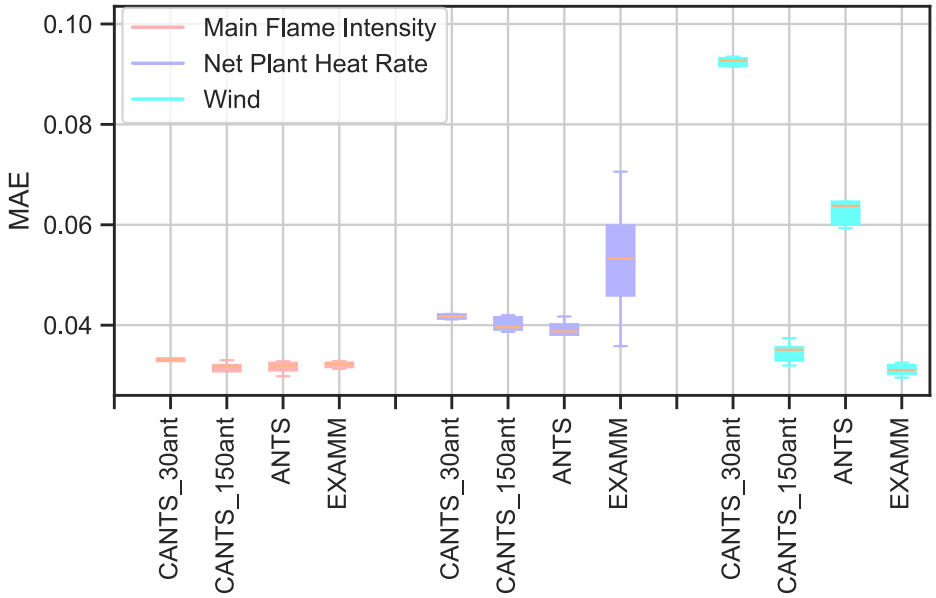


Figure 5.17: *Mean Average Error (MAE) ranges of best-found RNNs from each method.*

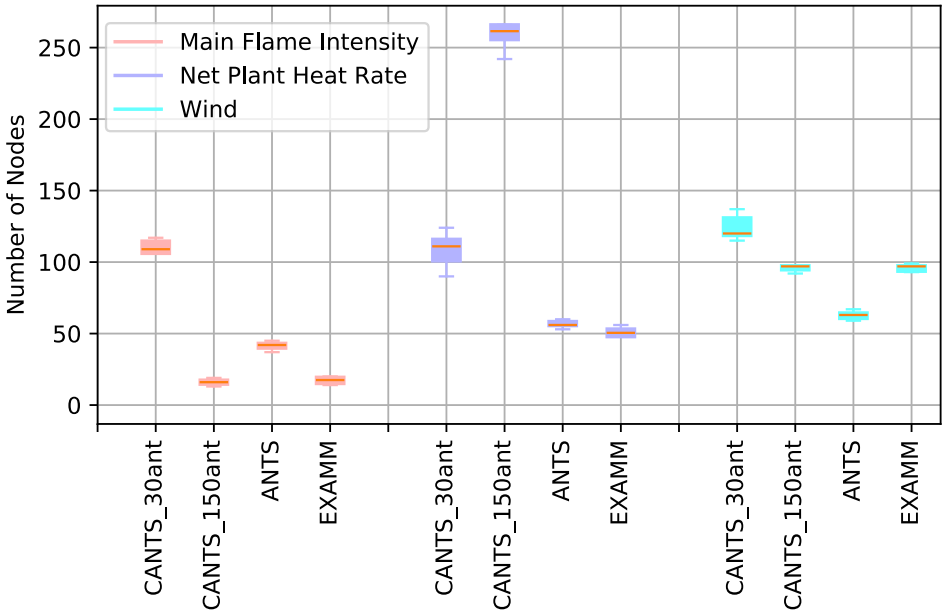


Figure 5.18: *Number of nodes in the best found RNNs from each method.*

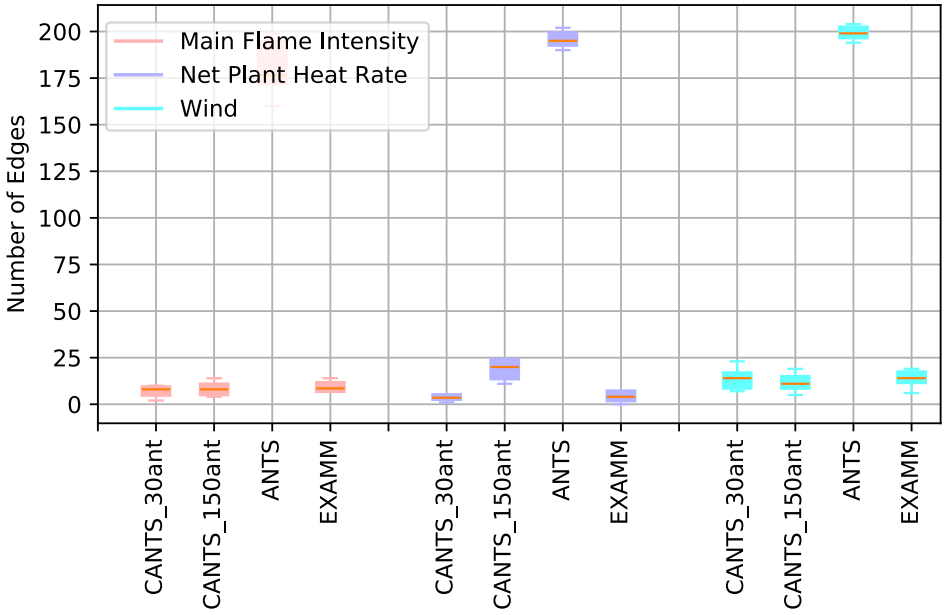


Figure 5.19: *Number of edges in the best-found RNNs from each algorithm.*

RNNs were each allowed 40 epochs of back-propagation for local fine-tuning (since all algorithms are mmetic). ANTS and EXAMM utilized the hyper-parameters previously reported to yield best results [40, 109].

The results shown in Figure 5.17, which compare CANTS, ANTS and EXAMM in the three experiments described above over the three datasets, report the range of mean average error (MAE) of each algorithm's best-found RNNs. While EXAMM outperformed CANTS with 30 ants, CANTS with 150 ants had a better performance than EXAMM and ANTS. CANTS was competitive with ANTS on the net plant heat rate predictions and outperformed EXAMM on this dataset. CANTS also outperformed ANTS on the wind energy dataset yet could not beat EXAMM. Potential reasons for this could be that the complexity/size of this dataset is greater and that the task is simply more difficult which results in a potentially larger search space. As CANTS allows for potentially unbounded network sizes, its search space is significantly larger than either that of ANTS or EXANM. Though ANTS outperformed CANTS on the wind dataset, CANTS is still a good competitor, especially since it has less hyper-parameters (8) to tune compared to both ANTS and EXAMM (both require at least 16). While all these reasons may be valid, the size of the search space is likely the biggest challenge. Further

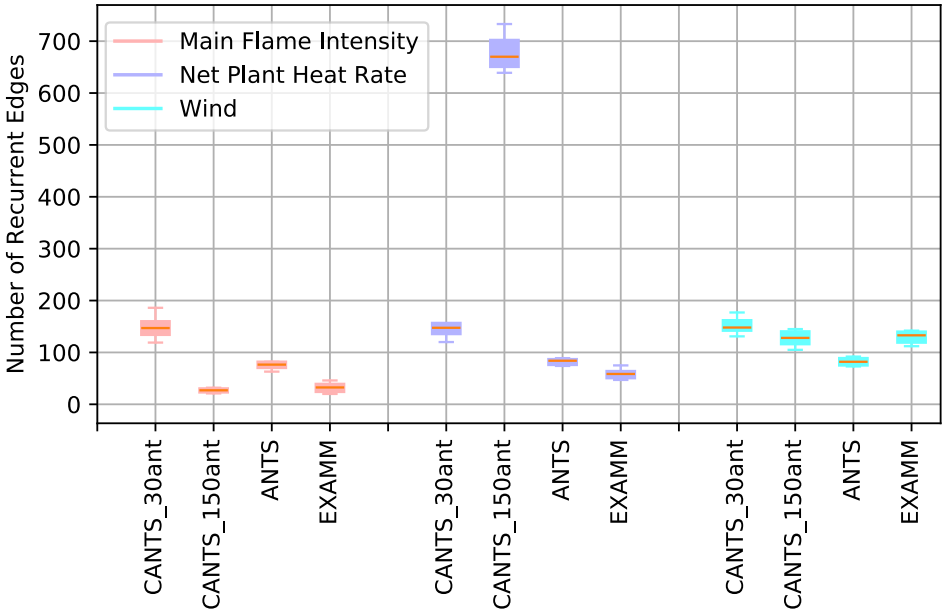


Figure 5.20: *Number of recurrent edges in the best-found RNNs each algorithm.*

evidence of this is provided in Figures 5.18, 5.19, 5.20, which present the number of structural elements (nodes, edges, and recurrent edges, respectively) of the best-found RNN architectures using the different algorithms. The CANTS runs with 150 ants resulted in significantly more complex architectures for many of the problems, which may be an indication that CANTS can evolve better performing structure if provided more optimization iterations.

Also, Figure 4.14 shows CANTS in action and how vast the continuous search space is and the endless possible structures which can be generated. The results in Figure 5.17 also show more consistency for CANTS over EXAMM with smaller box-plot confidence intervals and closer maximun and minimum value for the 10 repeats of the experiments. Figures 5.21- 5.27 show the RNNs at different iterations of optimization of process of CANTS while optimizing for Net Plant Heat Rate. From the figures it can be observed that the final RNN is deep and lost many of the input parameters.

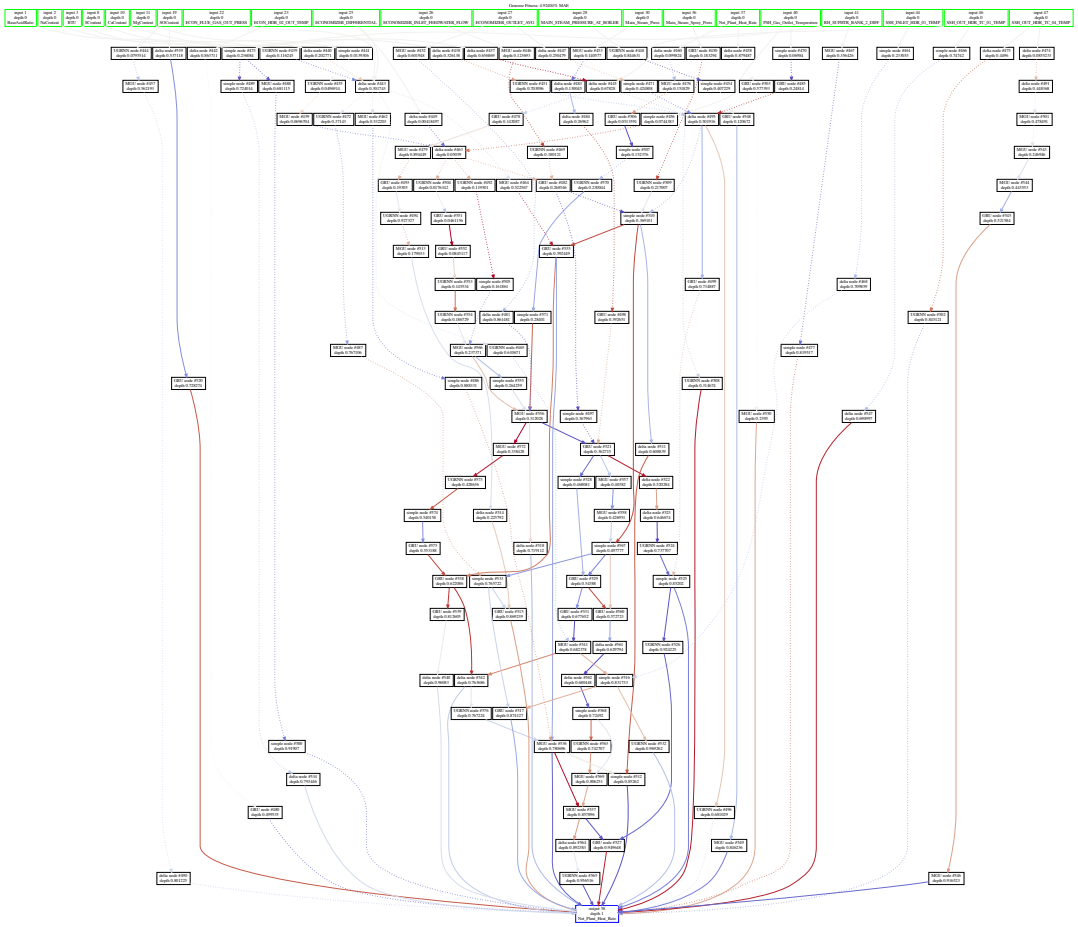


Figure 5.21: *Resulting Structure Iteration 1*

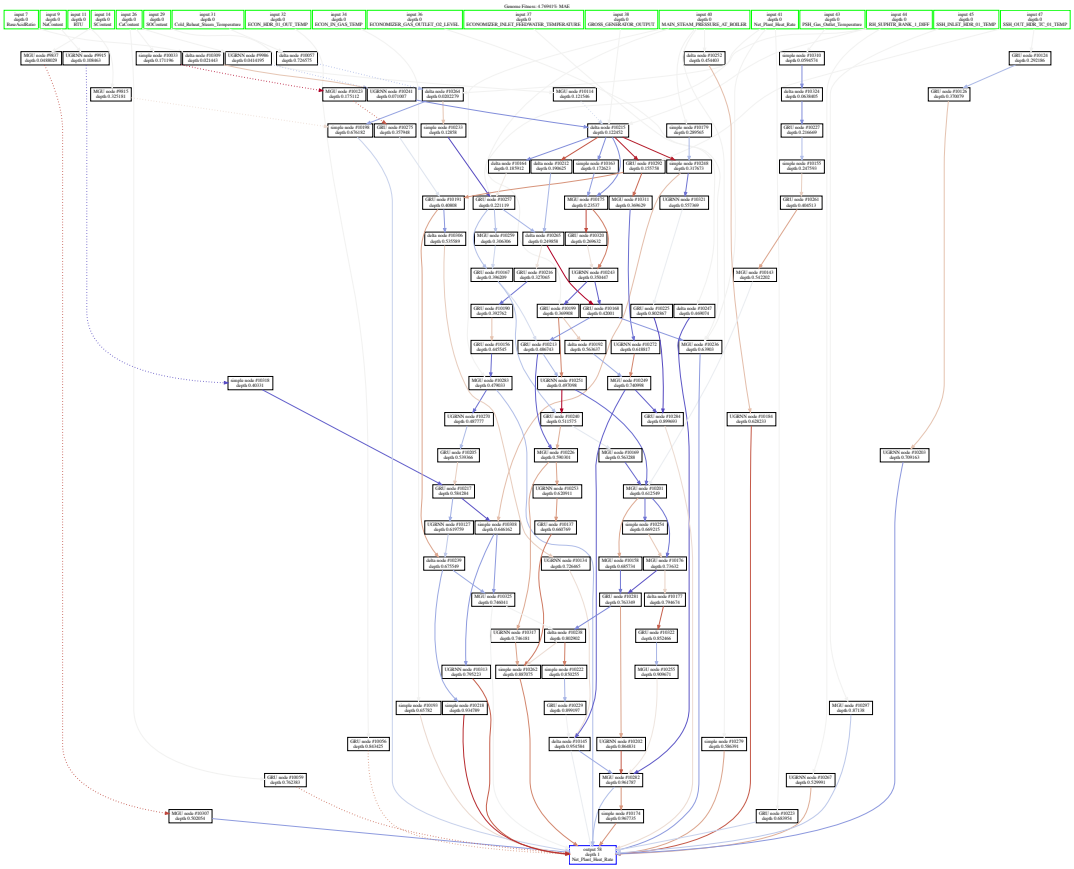


Figure 5.22: Resulting Structure Iteration 10

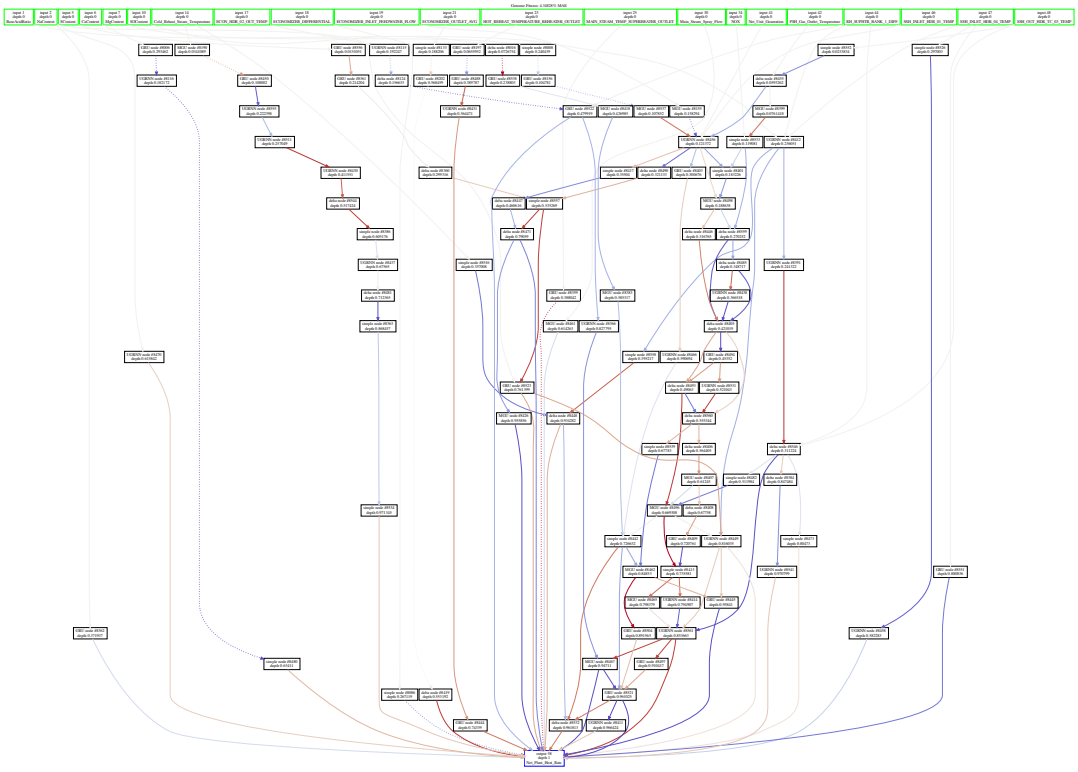


Figure 5.23: Resulting Structure Iteration 11

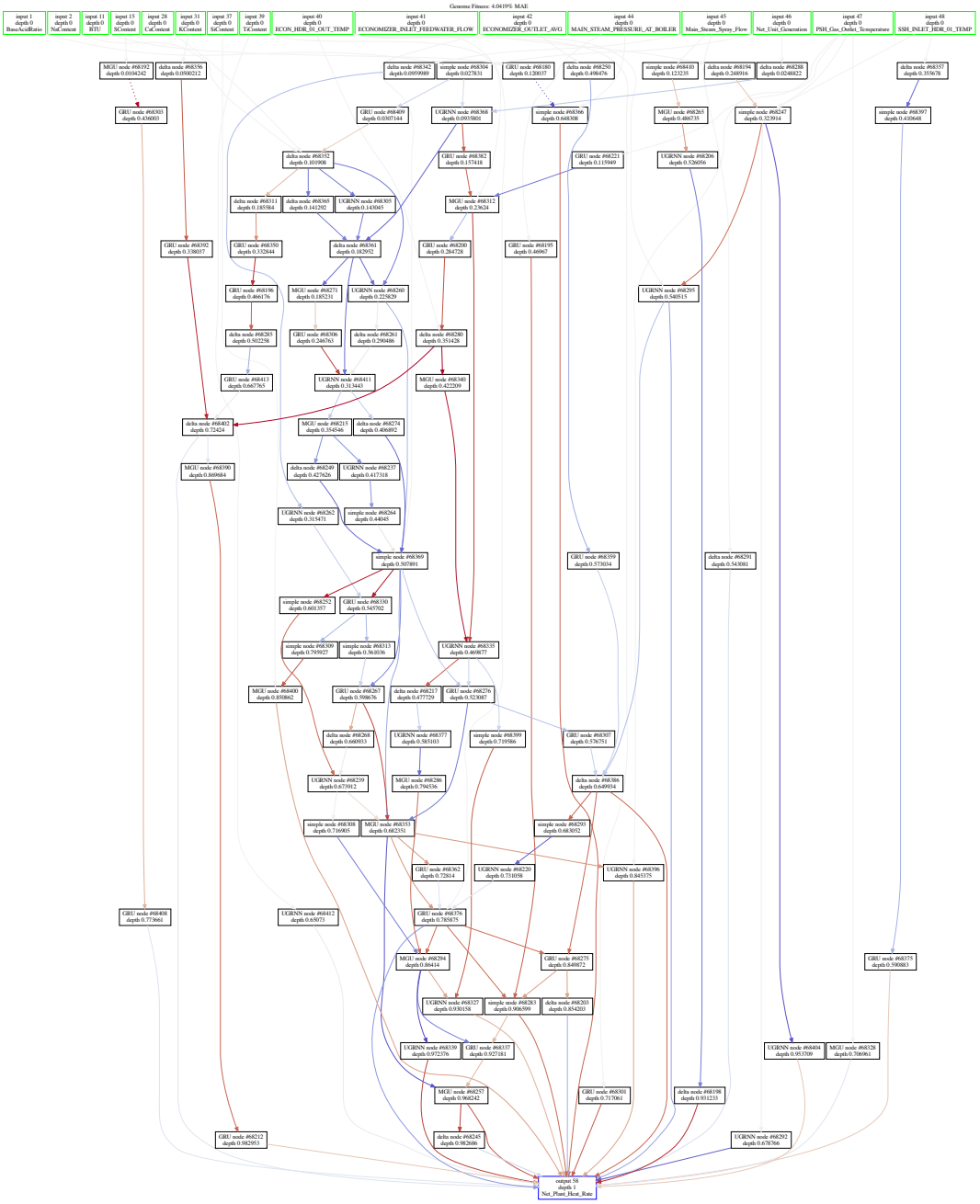


Figure 5.24: Resulting Structure Iteration 108

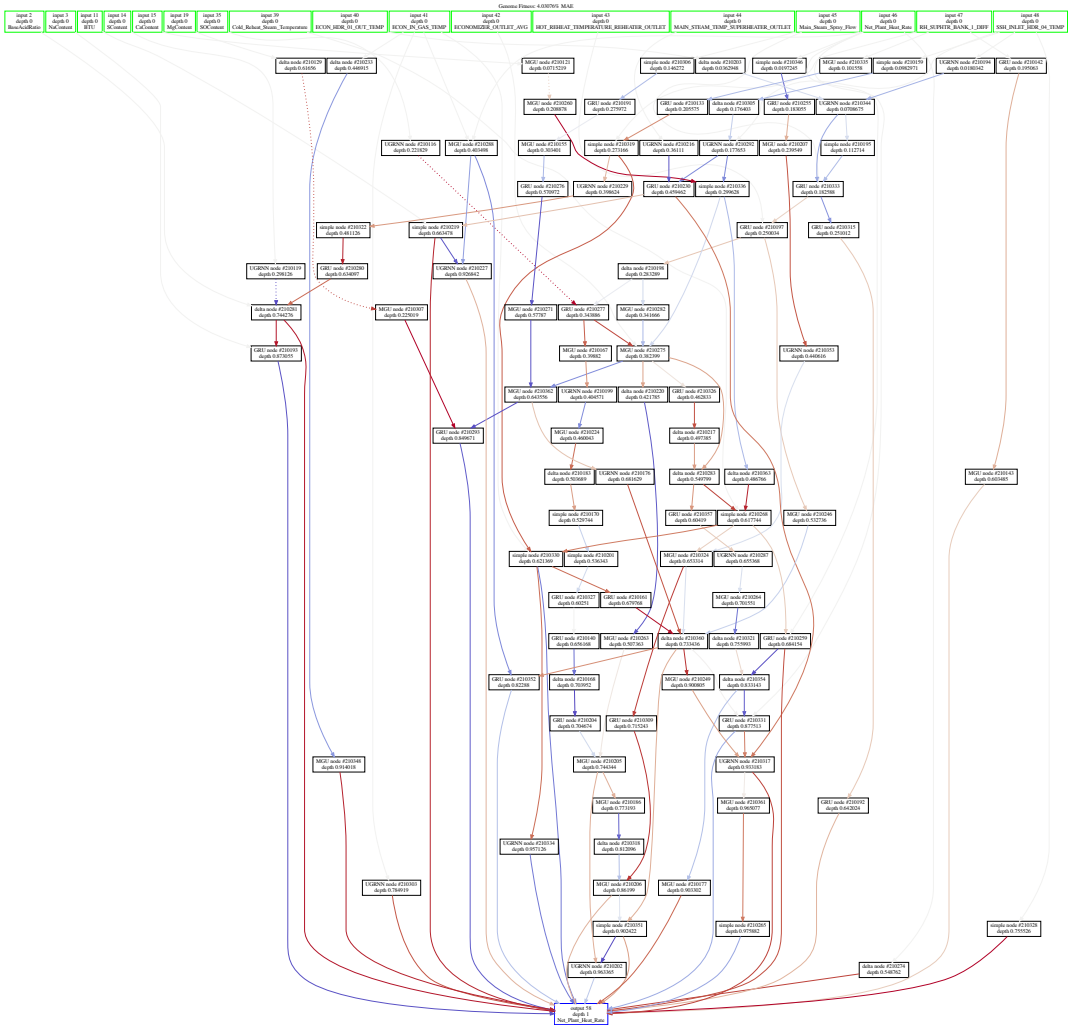


Figure 5.25: Resulting Structure Iteration 364

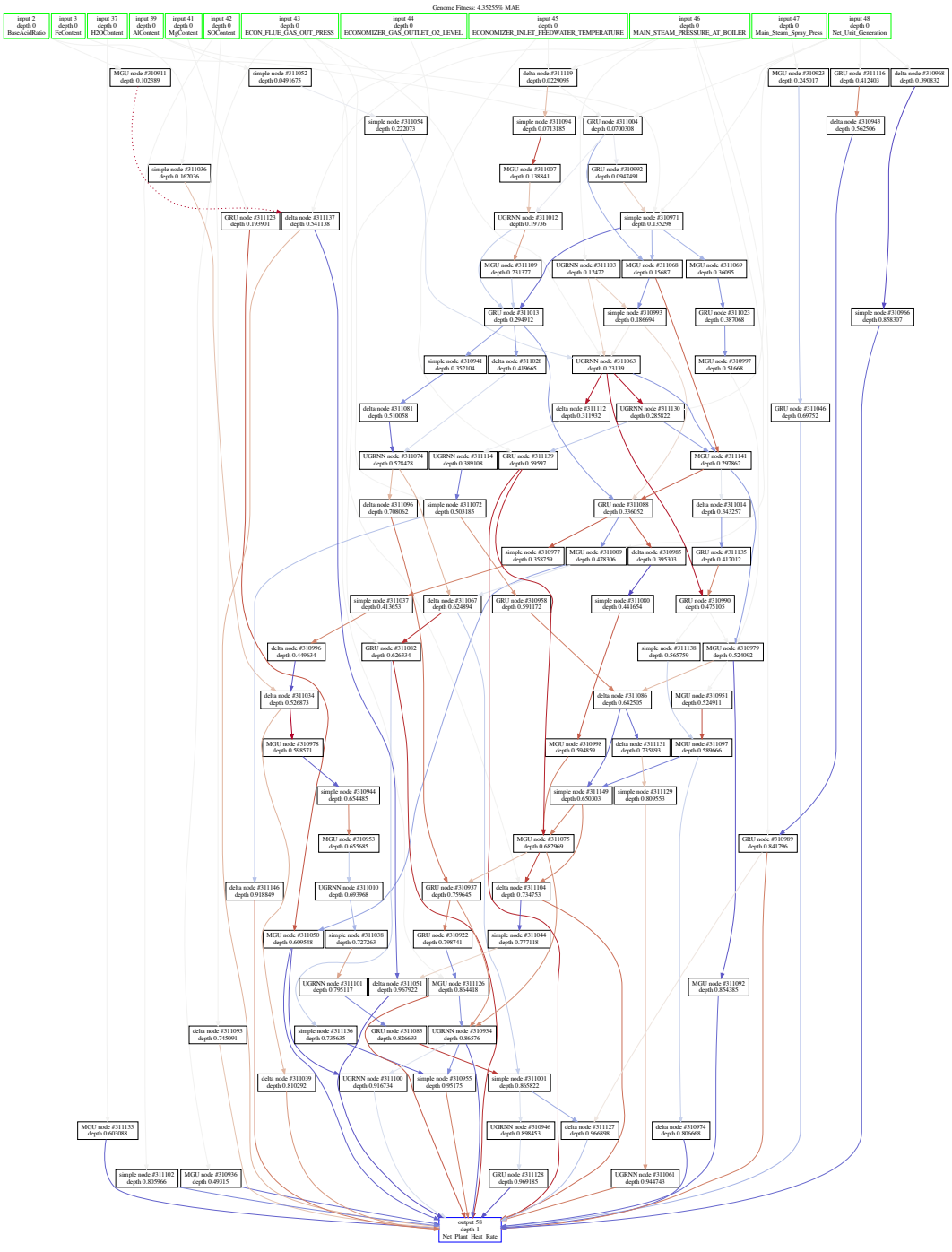


Figure 5.26: Resulting Structure Iteration 544

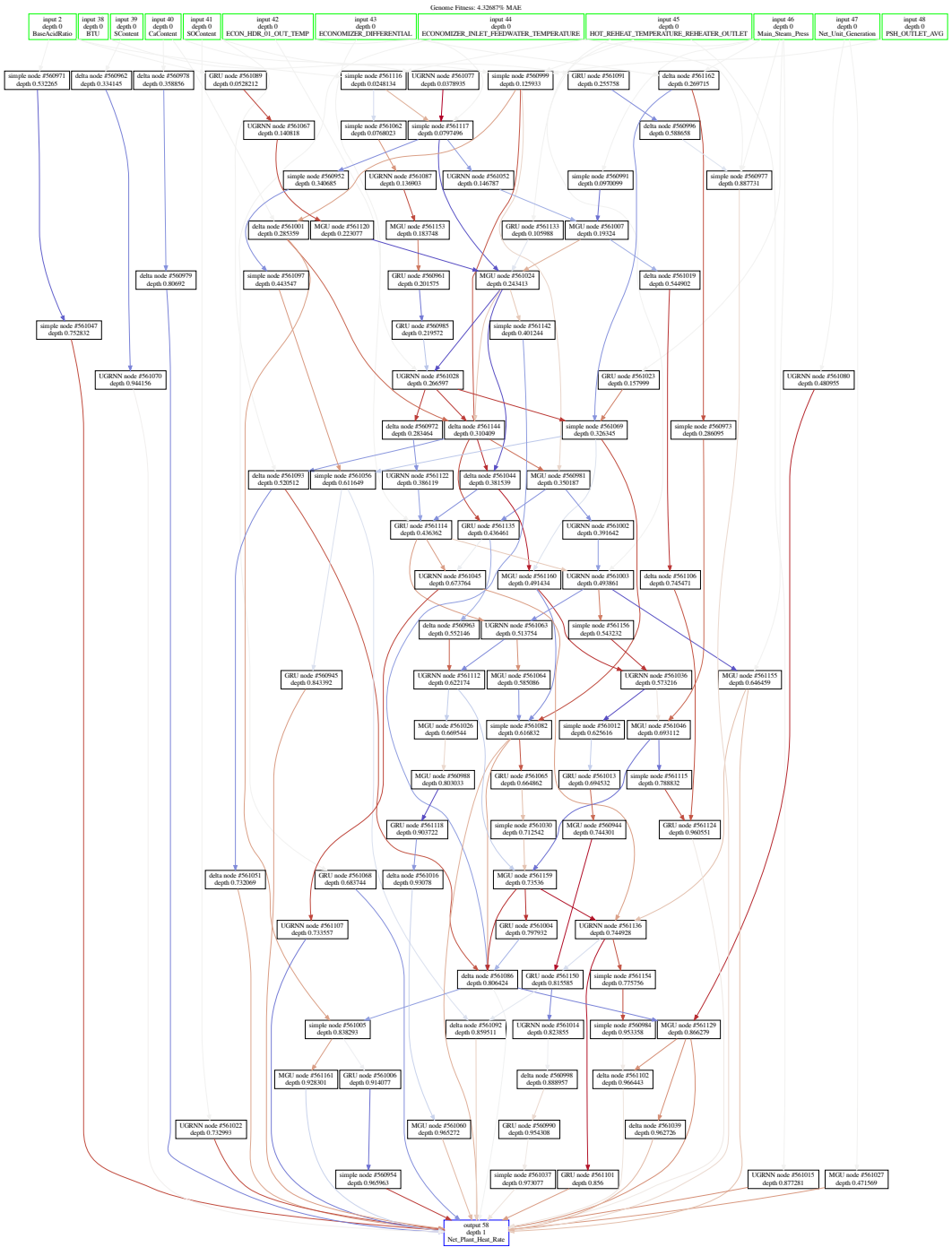


Figure 5.27: Resulting Structure Iteration 995

Chapter 6

Conclusion and Future Directions

This dissertation presents three key novel algorithms which are generalizations of the ant colony optimization (ACO) algorithm as applied to the field of neural architecture search (NAS). First, results from the memory cell-based ant colony optimization (MC-ACO) algorithm show that optimizing the gates within LSTM cells can dramatically reduce the number of connections required, while at the same time improve the predictive ability of the recurrent neural network. Second, to expand ACO to the problem of neuro-evolution/neural architecture search for recurrent neural networks (RNNs) with varying recurrent time spans and more complex memory cells, the Ant-based Neural Topology Search (ANTS) algorithm was developed. ANTS generates candidate RNNs from a massively-connected superstructure (the colony/swarm), taking advantage of ACO for structural optimization and concepts from neuro-evolutionary/genetic approaches for maintaining populations of RNN candidates that are trained locally and asynchronously, which makes ANTS a memetic procedure as well. A hallmark of ANTS is its computational formalization of role specialization as done by real ant colonies – ant agents are prevented from getting stuck "wandering" around the superstructure through the use of different ant roles, which are constrained to only explore different components of the underlying complex graph space. A novel communal intelligence strategy was also utilized for sharing and updating the best found weights within the colony. Third, ANTS' discrete search space was replaced with a continuous one in the Continuous ANTS (CANTS) algorithm, making the optimization agents closer to the ants in nature. The method is easier to use than ANTS with only eight user specified hyperparameters. CANTS also works asynchronously to accelerate the search process

while exploring its unbounded architectural search space.

MC-ACO yielded higher performance structures that outperformed regression-based model on a highly nonlinear dataset. Additionally, the results showed that MC-ACO outperformed the connection-dropout technique commonly used in neural network regularization.

For the ANTS algorithm, it was shown that using ant agents with different roles generated RNNs that were not only sparse but performant – these candidates almost entirely outperformed the more standard ant traversal strategies even when standard ants were biased to select forward paths. Additionally, communal weight sharing greatly improved the accuracy of the generated RNNs[1]. Moreover, allowing ants to jump (or skip) layers proved to not only boost performance but also to increase sparsity. Nonetheless, the introduction of L1 and L2 regularization into the ACO pheromone deposition process is quite novel, albeit a bit unconventional. The results show that by playing with the form of the pheromone adjustment function, the likelihood that sparser RNNs are found can be increased, which also can outperform schemes that do not incorporate regularization/constraints. The formalized strategies in this work are generic and could be applied to any other ACO algorithm's pheromone update process. The proposed ANTS metaheuristic not only provides advances and new concepts for the field of ant colony optimization research to further explore but also shows strong promise for its use as an alternative neuro-evolution algorithm for automated RNN architecture search. It significantly outperformed the well-known NEAT algorithm (even when NEAT was given an order of magnitude more computation). More importantly, ANTS outperforms the state-of-the-art EXAMM neuro-evolution algorithm on the studied time series problem.

Finally, ANTS was expanded to operate utilizing continuous search spaces, resulting in the CANTS algorithm. CANTS is an unorthodox generalization of the ACO algorithm in addition to the fact that it indirectly encodes neural architectural features into its constituent agents and their behavior when traversing the search space. CANTS allows ants to freely explore a limitless continuous search space without being restricted by a superstructure or a graph. This method has just eight parameters to tune: ants' sensing radii, the number of ants, the clustering number of points and proximity, number of time lags, pheromone evaporation and update parameters, and the ants' explo-

[1] Corroborating prior studies that have also shown the benefits of similar initialization schemes [27, 109].

ration/exploitation parameter, while ANTS has five heuristics to select from with 16 tuneable parameters. The results show that CANTS completes strongly with the other existing state-of-the-art methods with a powerful ability to explore an extremely vast search space with consistency.

The work underlying this thesis opens up multiple avenues for future study and presents several interesting questions. In particular, why were explorer ants able to find the best performing networks while performing quite poorly in the mean and median cases? Why did explorer ants combined with social recurrent ants perform extremely well in the mean and median cases but not in the best cases? Answering experimental questions such as these could lead to insights as to how recurrent connections that skip multiple steps of time interact with recurrent memory cells, potentially leading to the design of more expressive RNN structures that better capture longer-term dependencies in sequential data.

CANTS also opens a question regarding its parameter optimization and tuning – which requires careful consideration from the experimenter/user. To address this, future work will draw further inspiration from "lead ants" [60], which in nature act as scouts. Utilizing lead ants as another synthetic species could provide additional benefits to the exploration problem. Additionally, ant colonies can be expanded to evolve through a RL algorithm. The foraging ants can be generalized to learn from their own behaviour as individuals by comparing themselves to the group's behavior while they are doing their optimization work, which can help in improving the performance of the colony. Moreover, CANTS' ant agents will be extended to no longer conducting a discrete search between the (continuous plane) layers so that they can completely search a full 3D space in a continuous fashion, making the search completely continuous and potentially yielding even better neural architectures.

Furthermore, intra-colony direct communication between ants while they moving through the search space should be investigated and implemented. A potential hypothesis is that such a mechanism will let ants learn from their communal behaviour through a process that resembles reinforcement learning. Self-learning ant agents will make the colony evolve while optimizing the an architecture, bringing this algorithmic approach another step closer to their biological representations in nature. Additionally, following GA's concept of islands as niches for evolving different genome strides, and to further assimilate real ant colonies in nature, different colonies will be allowed to exist in parallel,

evolve architectures, and evolve themselves through crossovers between each other and through mutations.

It should be further noted that the generalizations proposed in this work and future work may even prove interesting for research conducted in Myrmecology[2]. It is possible to envision the synthetic ant colonies as living organisms with ants as their living cells [60], potentially offering a flexible, scalable simulation framework for modeling how ant agents might achieve more complex tasks related to overall survival.

Finally, future work will expand on the investigation of ANTS/CANTS on other time series datasets as well as sequence modeling (and classification) problems more commonly explored in mainstream statistical learning research, such as language modeling [101, 112].

[2]The branch of entomology focusing on the scientific study of ants.

Bibliography

[1] M Annunziato, M Lucchetti, and S Pizzuti. Adaptive systems and evolutionary neural networks: a survey. *Proc. EUNITE02, Albufeira, Portugal*, 2002.

[2] IA Azid, ZM Ripin, MS Aris, AL Ahmad, KN Seetharamu, and RM Yusoff. Predicting combined-cycle natural gas power plant emissions by using artificial neural networks. In *2000 TENCON Proceedings. Intelligent Systems and Technologies for the New Millennium (Cat. No. 00CH37119)*, volume 3, pages 512–517. IEEE, 2000.

[3] György Barna and Kimmo Kaski. *Choosing optimal network structure*, pages 890–893. Springer Netherlands, Dordrecht, 1990.

[4] Yoshua Bengio et al. Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1–127, 2009.

[5] Yoshua Bengio, Ian Goodfellow, and Aaron Courville. *Deep learning*, volume 1. Citeseer, 2017.

[6] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *The Journal of Machine Learning Research*, 13(1):281–305, 2012.

[7] Leonora Bianchi, Luca Maria Gambardella, and Marco Dorigo. An ant colony optimization approach to the probabilistic traveling salesman problem. In *International Conference on Parallel Problem Solving from Nature*, pages 883–892. Springer, 2002.

[8] George Bilchev and Ian C Parmee. The ant colony metaphor for searching continuous design spaces. In *AISB workshop on evolutionary computing*, pages 25–39. Springer, 1995.

[9] Christian Blum and Xiaodong Li. Swarm intelligence in optimization. In *Swarm Intelligence*, pages 43–85. Springer, 2008.

[10] Christian Blum and Krzysztof Socha. Training feed-forward neural networks with ant colony optimization: An application to pattern classification. In *Fifth International Conference on Hybrid Intelligent Systems (HIS'05)*, pages 6–pp. IEEE, 2005.

[11] Najmatoullahi Amadou Boukary. *A COMPARISON OF TIME SERIES FORECASTING LEARNING ALGORITHMS ON THE TASK OF PREDICTING EVENT TIMING*. PhD thesis, Royal Military College of Canada, 2016.

[12] Andrew Brock, Theodore Lim, James M Ritchie, and Nick Weston. Smash: one-shot model architecture search through hypernetworks. *arXiv preprint arXiv:1708.05344*, 2017.

[13] David S Broomhead and David Lowe. Radial basis functions, multi-variable functional interpolation and adaptive networks. Technical report, DTIC Document, 1988.

[14] Edvinas Byla and Wei Pang. Deepswarm: Optimising convolutional neural networks using swarm intelligence. *arXiv preprint arXiv:1905.07350*, 2019.

[15] Han Cai, Ligeng Zhu, and Song Han. Proxylessnas: Direct neural architecture search on target task and hardware. *arXiv preprint arXiv:1812.00332*, 2018.

[16] Chris Chatfield. *The analysis of time series: an introduction*. CRC press, 2016.

[17] Jun Cheng, Xin Wang, Tingting Si, Fan Zhou, Zhihua Wang, Junhu Zhou, and Kefa Cen. Maximum burning rate and fixed carbon burnout efficiency of power coal blends predicted with back-propagation neural network models. *Fuel*, 172:170–177, 2016.

[18] Jun Cheng, Xin Wang, Tingting Si, Fan Zhou, Junhu Zhou, and Kefa Cen. Ignition temperature and activation energy of power coal blends predicted with back-propagation neural network models. *Fuel*, 173:230–238, 2016.

[19] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.

[20] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[21] David A Clifton, Peter R Bannister, and Lionel Tarassenko. A framework for novelty detection in jet engine vibration data. In *Key engineering materials*, volume 347, pages 305–310. Trans Tech Publ, 2007.

[22] Jasmine Collins, Jascha Sohl-Dickstein, and David Sussillo. Capacity and trainability in recurrent neural networks. *arXiv preprint arXiv:1611.09913*, 2016.

[23] Lisandro Dalcín, Rodrigo Paz, Mario Storti, and Jorge D'Elía. Mpi for python: Performance improvements and mpi-2 extensions. *Journal of Parallel and Distributed Computing*, 68(5):655–662, 2008.

[24] Ashraf Darwish, Aboul Ella Hassanien, and Swagatam Das. A survey of swarm and evolutionary computing approaches for deep learning. *Artificial Intelligence Review*, 53(3):1767–1812, 2020.

[25] S De, Mehrzad Kaiadi, Magnus Fast, and Mohsen Assadi. Development of an artificial neural network model for the steam process of a coal biomass cofired combined heat and power (chp) plant in sweden. *Energy*, 32(11):2099–2109, 2007.

[26] Travis Desell. Large scale evolution of convolutional neural networks using volunteer computing. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 127–128. ACM, 2017.

[27] Travis Desell. Accelerating the evolution of convolutional neural networks with node-level mutations and epigenetic weight initialization. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 157–158. ACM, 2018.

[28] Travis Desell, Sophine Clachar, James Higgins, and Brandon Wild. Evolving neural network weights for time-series prediction of general aviation flight data. In Thomas Bartz-Beielstein, Jürgen Branke, Bogdan Filipič, and Jim Smith, editors, *Parallel Problem Solving from Nature - PPSN XIII*, volume 8672 of *Lecture Notes in Computer Science*, pages 771–781. Springer International Publishing, 2014.

[29] Travis Desell, Sophine Clachar, James Higgins, and Brandon Wild. Evolving deep recurrent neural networks using ant colony optimization. In Gabriela Ochoa

and Francisco Chicano, editors, *Evolutionary Computation in Combinatorial Optimization*, pages 86–98, Cham, 2015. Springer International Publishing.

[30] Marco Dorigo. Optimization, learning and natural algorithms. *PhD Thesis, Politecnico di Milano*, 1992.

[31] Marco Dorigo, Mauro Birattari, and Thomas Stutzle. Ant colony optimization. *IEEE computational intelligence magazine*, 1(4):28–39, 2006.

[32] Marco Dorigo, Mauro Birattari, and Thomas Stutzle. Ant colony optimization. *IEEE computational intelligence magazine*, 1(4):28–39, 2006.

[33] Marco Dorigo and Luca Maria Gambardella. Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Transactions on evolutionary computation*, 1(1):53–66, 1997.

[34] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. Ant system: optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 26(1):29–41, 1996.

[35] Marco Dorigo and Thomas Stützle. Ant colony optimization: overview and recent advances. In *Handbook of metaheuristics*, pages 227–263. Springer, 2010.

[36] Johann Dréo and Patrick Siarry. A new ant colony algorithm using the heterarchical concept aimed at optimization of multiminima continuous functions. In *International Workshop on Ant Algorithms*, pages 216–221. Springer, 2002.

[37] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

[38] AbdElRahman ElSaid, Steven Benson, Shuchita Patwardhan, David Stadem, and Travis Desell. Evolving recurrent neural networks for time series data prediction of coal plant parameters. In *International Conference on the Applications of Evolutionary Computation (Part of EvoStar)*, pages 488–503. Springer, 2019.

[39] AbdElRahman ElSaid, Fatima El Jamiy, James Higgins, Brandon Wild, and Travis Desell. Optimizing long short-term memory recurrent neural networks using ant colony optimization to predict turbine engine vibration. *Applied Soft Computing*, 73:969–991, 2018.

[40] AbdElRahman ElSaid, Alexander G Ororbia, and Travis J Desell. Ant-based neural topology search (ants) for optimizing recurrent networks. In *International Conference on the Applications of Evolutionary Computation (Part of EvoStar)*, pages 626–641. Springer, 2020.

[41] AbdElRahman ElSaid, Brandon Wild, James Higgins, and Travis Desell. Using lstm recurrent neural networks to predict excess vibration events in aircraft engines. In *e-Science (e-Science), 2016 IEEE 12th International Conference on*, pages 260–269. IEEE, 2016.

[42] AbdElRahman ElSaid, Brandon Wild, Fatima El Jamiy, James Higgins, and Travis Desell. Optimizing lstm rnns using aco to predict turbine engine vibration. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 21–22. ACM, 2017.

[43] AbdElRahman A. ElSaid, Alexander G. Ororbia, and Travis J. Desell. The ant swarm neuro-evolution procedure for optimizing recurrent networks, 2019.

[44] Thomas Elsken, Jan-Hendrik Metzen, and Frank Hutter. Simple and efficient architecture search for convolutional neural networks. *arXiv preprint arXiv:1711.04528*, 2017.

[45] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Efficient multi-objective neural architecture search via lamarckian evolution. *arXiv preprint arXiv:1804.09081*, 2018.

[46] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *arXiv preprint arXiv:1808.05377*, 2018.

[47] Okan Erkaymaz, Mahmut Özer, and Nejat Yumuşak. Impact of small-world topology on the performance of a feed-forward artificial neural network based on 2 different real-life problems. *Turkish Journal of Electrical Engineering & Computer Sciences*, 22(3):708–718, 2014.

[48] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

[49] Felix A. Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to Forget: Continual Prediction with LSTM. *Neural Computation, Vol. 12, No. 10 , Pages 2451-2471*, October 2000.

[50] Ben Fielding and Li Zhang. Evolving image classification architectures with enhanced particle swarm optimisation. *IEEE Access*, 6:68560–68575, 2018.

[51] Iztok Fister Jr, Xin-She Yang, Iztok Fister, Janez Brest, and Dušan Fister. A brief review of nature-inspired algorithms for optimization. *arXiv preprint arXiv:1307.4186*, 2013.

[52] Edgar Galván and Peter Mooney. Neuroevolution in deep neural networks: Current trends and future challenges. *arXiv preprint arXiv:2006.05415*, 2020.

[53] Jason Gauci and Kenneth O Stanley. Autonomous evolution of topographic regularities in artificial neural networks. *Neural computation*, 22(7):1860–1898, 2010.

[54] Felix A Gers, Douglas Eck, and Jürgen Schmidhuber. Applying lstm to time series predictable through time-window approaches. In *Neural Nets WIRN Vietri-01*, pages 193–200. Springer, 2002.

[55] Felix A Gers, Nicol N Schraudolph, and Jürgen Schmidhuber. Learning precise timing with lstm recurrent networks. *Journal of machine learning research*, 3(Aug):115–143, 2002.

[56] Vilas N Ghate and Sanjay V Dudul. Cascade neural-network-based fault classifier for three-phase induction motor. *IEEE Transactions on Industrial Electronics*, 58(5):1555–1563, 2011.

[57] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[58] Hardik Goel, Igor Melnyk, Nikunj C. Oza, Bryan Matthews, and Arindam Banerjee. Multivariate aviation time series modeling : Vars vs . lstms. 2016.

[59] Xinyu Gong, Shiyu Chang, Yifan Jiang, and Zhangyang Wang. Autogan: Neural architecture search for generative adversarial networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 3224–3234, 2019.

[60] Deborah M Gordon. *Ant encounters: interaction networks and colony behavior*, volume 1. Princeton University Press, 2010.

[61] Himanshu Gupta and Bahniman Ghosh. Transistor size optimization in digital circuits using ant colony optimization for continuous domain. *International Journal of Circuit Theory and Applications*, 42(6):642–658, 2014.

[62] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[63] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116, 1998.

[64] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, Jürgen Schmidhuber, et al. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001.

[65] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[66] Ming-Huwi Horng. Fine-tuning parameters of deep belief networks using artificial bee colony algorithm. *DEStech Transactions on Computer Science and Engineering*, (aita), 2017.

[67] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[68] Christian Igel. Neuroevolution for reinforcement learning using evolution strategies. In *The 2003 Congress on Evolutionary Computation, 2003. CEC'03.*, volume 4, pages 2588–2595. IEEE, 2003.

[69] J.-B. Li and Y.-K. Chung. A novel back-propagation neural network training algorithm designed by an ant colony optimization. *In Transmission and Distribution Conference and Exhibition: Asia and Pacific, 2005 IEEE/PES, pages 1–5. IEEE*, 2005.

[70] Yesmina Jaafra, Jean Luc Laurent, Aline Deruyver, and Mohamed Saber Naceur. Reinforcement learning for neural architecture search: A review. *Image and Vision Computing*, 89:57–66, 2019.

[71] Herbert Jaeger. Echo state network. *Scholarpedia*, 2(9):2330, 2007.

[72] Michael I Jordan. Serial order: A parallel distributed processing approach. In *Advances in psychology*, volume 121, pages 471–495. Elsevier, 1997.

[73] E Jorjani, S Chehreh Chelgani, and Sh Mesroghli. Prediction of microbial desulfurization of coal using artificial neural networks. *Minerals Engineering*, 20(14):1285–1292, 2007.

[74] E Jorjani, S Chehreh Chelgani, and SH Mesroghli. Application of artificial neural networks to predict chemical desulfurization of tabas coal. *Fuel*, 87(12):2727–2734, 2008.

[75] Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. An empirical exploration of recurrent network architectures. In *International Conference on Machine Learning*, pages 2342–2350, 2015.

[76] K. Socha. Aco for continuous and mixed-variable optimization. in ant colony optimization and swarm intelligence. *Future Generation Computer Systems*, pages 25—-36, 2004.

[77] K. Sochaand & M. Dorigo. Bottom hole pressure estimation using evolved neural networks by real coded ant colony optimization and genetic algorithm. *Journal of Petroleum Science and Engineering, 77(3):375–385*, 2011.

[78] Kirthevasan Kandasamy, Willie Neiswanger, Jeff Schneider, Barnabas Poczos, and Eric P Xing. Neural architecture search with bayesian optimisation and optimal transport. In *Advances in neural information processing systems*, pages 2016–2025, 2018.

[79] Eric R Kandel, James H Schwartz, Thomas M Jessell, Steven A Siegelbaum, and A James Hudspeth. *Principles of neural science*, volume 4. McGraw-hill New York, 2000.

[80] Hideki Katagiri, Ichiro Nishizaki, Tomohiro Hayashida, and Takanori Kadoma. Multiobjective evolutionary optimization of training and topology of recurrent

neural networks for time-series prediction. *The Computer Journal*, 55(3):325–336, 2011.

[81] James Kennedy. Particle swarm optimization. In *Encyclopedia of machine learning*, pages 760–766. Springer, 2011.

[82] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[83] Lachlan D Kuhn. Ant colony optimization for continuous spaces. 2002.

[84] Amrita Kumari, SK Das, and PK Srivastava. Modeling fireside corrosion rate in a coal fired boiler using adaptive neural network formalism. *Portugaliae Electrochimica Acta*, 34(1):23–38, 2016.

[85] Hugo Larochelle, Yoshua Bengio, Jérôme Louradour, and Pascal Lamblin. Exploring strategies for training deep neural networks. *Journal of Machine Learning Research*, 10(Jan):1–40, 2009.

[86] Ryno Laubscher. Time-series forecasting of coal-fired power plant reheater metal temperatures using encoder-decoder recurrent neural networks. *Energy*, 189:116187, 2019.

[87] Collins Leke, Alain Richard Ndjiongue, Bhekisipho Twala, and Tshilidzi Marwala. A deep learning-cuckoo search method for missing data estimation in high-dimensional datasets. In *International Conference on Swarm Intelligence*, pages 561–572. Springer, 2017.

[88] David Leverington. A basic introduction to feedforward backpropagation neural networks. *Neural Network Basics*, 2012.

[89] Kang Li, Steve Thompson, and Jianxun Peng. Modelling and prediction of nox emission in a coal-fired power generation plant. *Control Engineering Practice*, 12(6):707–723, 2004.

[90] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

[91] XJ Liu, XB Kong, GL Hou, and JH Wang. Modeling of a 1000mw power plant ultra super-critical boiler system using fuzzy-neural network methods. *Energy Conversion and Management*, 65:518–527, 2013.

[92] Yan-Peng Liu, Ming-Guang Wu, and Ji-Xin Qian. Evolving neural networks using the hybrid of ant colony optimization and bp algorithms. In *International Symposium on Neural Networks*, pages 714–722. Springer, 2006.

[93] Y.P. Liu, M.G. Wu, and J.X. Qian. Predicting coal ash fusion temperature based on its chemical composition using aco-bp neural network. *Thermochimica Acta*, 454(1):64 – 68, 2007.

[94] Yuqiao Liu, Yanan Sun, Bing Xue, Mengjie Zhang, and Gary Yen. A survey on evolutionary neural architecture search. *arXiv preprint arXiv:2008.10937*, 2020.

[95] You Lv, Jizhen Liu, Tingting Yang, and Deliang Zeng. A novel least squares support vector machine ensemble model for nox emission prediction of a coal-fired boiler. *Energy*, 55:319–329, 2013.

[96] M. Dorigo and L. M. Gambardella. Ant colonies for the travelling sales man problem. *BioSystems, 43(2):73–81*, 1997.

[97] M. Unal, M. Onat, and A. Bal. Cellular neural network training by ant colony optimization algorithm. *In Signal Processing and Communications Applications Conference (SIU), 2010 IEEE 18th, pages 471–474. IEEE*, 2010.

[98] Max Manfrin, Mauro Birattari, Thomas Stützle, and Marco Dorigo. Parallel ant colony optimization for the traveling salesman problem. In *International Workshop on Ant Colony Optimization and Swarm Intelligence*, pages 224–234. Springer, 2006.

[99] Adam H Marblestone, Greg Wayne, and Konrad P Kording. Toward an integration of deep learning and neuroscience. *Frontiers in computational neuroscience*, 10:94, 2016.

[100] Michalis Mavrovouniotis and Shengxiang Yang. Evolving neural networks using ant colony optimization with pheromone trail limits. In *Computational Intelligence (UKCI), 2013 13th UK Workshop on*, pages 16–23. IEEE, 2013.

[101] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Eleventh annual conference of the international speech communication association*, 2010.

[102] Michael C Mozer. A focused backpropagation algorithm for temporal. *Backpropagation: Theory, architectures, and applications*, 137, 1995.

[103] N. Monmarché and G. Venturini and M. Slimane. On how pachycondyla apicalis ants suggest a new search algorithm. *Future Generation Computer Systems*, 16:937–946, 2000.

[104] Alexandre Nairac, Neil Townsend, Roy Carr, Steve King, Peter Cowley, and Lionel Tarassenko. A system for the analysis of jet engine vibration data. *Integrated Computer-Aided Engineering*, 6(1):53–66, 1999.

[105] Renato Negrinho and Geoff Gordon. Deeparchitect: Automatically designing and training deep architectures. *arXiv preprint arXiv:1704.08792*, 2017.

[106] Oliver Nelles. *Nonlinear system identification: from classical approaches to neural networks and fuzzy models*. Springer Science & Business Media, 2013.

[107] Michael A Nielsen. Neural networks and deep learning. *http://neuralnetworksanddeeplearning.com*, 2015.

[108] Cem Onat and Mahmut Daskin. A basic ann system for prediction of excess air coefficient on coal burners equipped with a ccd camera. *Mathematics and Statistics*, 7(1):1–9, 2019.

[109] Alexander Ororbia, AbdElRahman ElSaid, and Travis Desell. Investigating recurrent neural network memory structures using neuro-evolution. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '19, pages 446–455, New York, NY, USA, 2019. ACM.

[110] Alexander G Ororbia and Ankur Mali. Biologically motivated algorithms for propagating local target representations. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 4651–4658, 2019.

[111] Alexander G Ororbia, Ankur Mali, Daniel Kifer, and C Lee Giles. Conducting credit assignment by aligning local representations. *arXiv preprint arXiv:1803.01834*, 2018.

[112] Alexander G Ororbia II, Tomas Mikolov, and David Reitter. Learning simpler language models with the differential state framework. *Neural computation*, 29(12):3327–3352, 2017.

[113] Sean O'Donnell, Susan Bulova, Meghan Barrett, and Christoph von Beeren. Brain investment under colony-level selection: soldier specialization in eciton army ants (formicidae: Dorylinae). *BMC Zoology*, 3(1):3, 2018.

[114] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, pages 1310–1318, 2013.

[115] Joshua Peake, Martyn Amos, Paraskevas Yiapanis, and Huw Lloyd. Scaling techniques for parallel ant colony optimization on large problem instances. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 47–54, 2019.

[116] Hieu Pham, Melody Y Guan, Barret Zoph, Quoc V Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268*, 2018.

[117] Riccardo Poli, James Kennedy, and Tim Blackwell. Particle swarm optimization. *Swarm intelligence*, 1(1):33–57, 2007.

[118] Prangya Parimita Pradhan and Bidyadhar Subudhi. Wind speed forecasting based on wavelet transformation and recurrent neural network. *International Journal of Numerical Modelling: Electronic Networks, Devices and Fields*, 33(1):e2670, 2020.

[119] Jonas Prellberg and Oliver Kramer. Lamarckian evolution of convolutional neural networks. In *International Conference on Parallel Problem Solving from Nature*, pages 424–435. Springer, 2018.

[120] R. Ashena and J. Moghadasi. Ant colony optimization for continuous domains. *European journal of operational research, 185(3):1155–1173*, 2008.

[121] Gireesh S. S. Raman and Mark S. Klima. Application of artificial neural networks for evaluating pressure filtration of coal refuse slurries. *Mineral Processing and Extractive Metallurgy Review*, 38(1):47–53, 2017.

[122] Vesna M Ranković and Ilija Ž Nikolić. Identification of nonlinear models with feed forward neural network and digital recurrent network. *FME Transactions*, 36(2):87–92, 2008.

[123] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc Le, and Alex Kurakin. Large-scale evolution of image classifiers. *arXiv preprint arXiv:1703.01041*, 2017.

[124] Pengzhen Ren, Yun Xiao, Xiaojun Chang, Po-Yao Huang, Zhihui Li, Xiaojiang Chen, and Xin Wang. A comprehensive survey of neural architecture search: Challenges and solutions. *arXiv preprint arXiv:2006.02903*, 2020.

[125] Luis Miguel Rios and Nikolaos V Sahinidis. Derivative-free optimization: a review of algorithms and comparison of software implementations. *Journal of Global Optimization*, 56(3):1247–1293, 2013.

[126] AJ Robinson and Frank Fallside. *The utility driven dynamic error propagation network*. University of Cambridge Department of Engineering Cambridge, 1987.

[127] David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

[128] S. Hochrieter & J. Schmidhuber. Long Short Term Memory. *Neural Computation 9(8):1735-1780*, 1997.

[129] KS Sandhu, Anil Ramachandran Nair, et al. A comparative study of arima and rnn for short term wind speed forecasting. In *2019 10th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, pages 1–7. IEEE, 2019.

[130] Tomonobu Senjyu, Atsushi Yona, Naomitsu Urasaki, and Toshihisa Funabashi. Application of recurrent neural network to long-term-ahead generating power forecasting for wind power generator. In *2006 IEEE PES Power Systems Conference and Exposition*, pages 1260–1265. IEEE, 2006.

[131] Rahul Karthik Sivagaminathan and Sreeram Ramakrishnan. A hybrid approach for feature subset selection using neural networks and ant colony optimization. *Expert systems with applications*, 33(1):49–60, 2007.

[132] J Smrekar, D Pandit, Magnus Fast, Mohsen Assadi, and Sudipta De. Prediction of power output of a coal-fired power plant by artificial neural network. *Neural Computing and Applications*, 19(5):725–740, 2010.

[133] Krzysztof Socha. *Ant colony optimisation for continuous and mixed-variable domains*. VDM Publishing Saarbrücken, 2009.

[134] Krzysztof Socha and Marco Dorigo. Ant colony optimization for continuous domains. *European journal of operational research*, 185(3):1155–1173, 2008.

[135] Robert Sowa et al. *Untersuchung von Synchronisationsphänomenen in dynamischen Systemen mit zellularen neuronalen Netzen*. PhD thesis, University of Bonn, Germany, 2004.

[136] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[137] Tushar Srivastava, Vedanshu, and MM Tripathi. Predictive analysis of rnn, gbm and lstm network for short-term wind power forecasting. *Journal of Statistics and Management Systems*, 23(1):33–47, 2020.

[138] Kenneth O Stanley, Jeff Clune, Joel Lehman, and Risto Miikkulainen. Designing neural networks through neuroevolution. *Nature Machine Intelligence*, 1(1):24–35, 2019.

[139] Kenneth O Stanley, David B D'Ambrosio, and Jason Gauci. A hypercube-based encoding for evolving large-scale neural networks. *Artificial life*, 15(2):185–212, 2009.

[140] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[141] Rainer Storn and Kenneth Price. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, 11(4):341–359, 1997.

[142] Yanan Sun, Gary G Yen, and Zhang Yi. Evolving unsupervised deep neural networks for learning meaningful representations. *IEEE Transactions on Evolutionary Computation*, 23(1):89–103, 2018.

[143] Peng Tan, Biao He, Cheng Zhang, Debei Rao, Shengnan Li, Qingyan Fang, and Gang Chen. Dynamic modeling of nox emission in a 660 mw coal-fired boiler with long short-term memory. *Energy*, 176:429–436, 2019.

[144] KS Tang, CY Chan, KF Man, and S Kwong. Genetic structure for nn topology and weights optimization. 1995.

[145] Enrique Teruel, Cristóbal Cortés, Luis Ignacio Díez, and Inmaculada Arauzo. Monitoring and prediction of fouling in coal-fired utility boilers using neural networks. *Chemical Engineering Science*, 60(18):5035 – 5048, 2005.

[146] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[147] Yasin Tunckaya and Etem Koklukaya. Comparative prediction analysis of 600 mwe coal-fired power plant production rate using statistical and neural-based models. *Journal of the Energy Institute*, 88(1):11–18, 2015.

[148] Bin Wang, Yanan Sun, Bing Xue, and Mengjie Zhang. Evolving deep convolutional neural networks by variable-length particle swarm optimization for image classification. In *2018 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8. IEEE, 2018.

[149] Bin Wang, Yanan Sun, Bing Xue, and Mengjie Zhang. A hybrid differential evolution approach to designing deep convolutional neural networks for image classification. In *Australasian Joint Conference on Artificial Intelligence*, pages 237–250. Springer, 2018.

[150] Junchao Wang, Weidong Fan, Yu Li, Meng Xiao, Kang Wang, and Peng Ren. The effect of air staged combustion on nox emissions in dried lignite combustion. *Energy*, 37(1):725–736, 2012.

[151] Xiaoxia Wang, Liangyu Ma, Bingshu Wang, and Tao Wang. A hybrid optimization-based recurrent neural network for real-time data prediction. *Neurocomputing*, 120:547 – 559, 2013. Image Feature Detection and Description.

[152] Yan Wang, Xuelei Sherry Ni, and Brian Stone. A two-stage hybrid model by using artificial neural networks as feature construction algorithms. *International Journal of Data Mining & Knowledge Management Process (IJDKP) Vol*, 8, 2018.

[153] Paul J Werbos. Generalization of backpropagation with application to a recurrent gas market model. *Neural networks*, 1(4):339–356, 1988.

[154] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[155] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[156] Martin Wistuba, Ambrish Rawat, and Tejaswini Pedapati. A survey on neural architecture search. *arXiv preprint arXiv:1905.01392*, 2019.

[157] Jing Xiao and LiangPing Li. A hybrid ant colony optimization for continuous domains. *Expert Systems with Applications*, 38(9):11072–11077, 2011.

[158] Sirui Xie, Hehui Zheng, Chunxiao Liu, and Liang Lin. Snas: stochastic neural architecture search. *arXiv preprint arXiv:1812.09926*, 2018.

[159] Xin-She Yang. *Nature-inspired metaheuristic algorithms*. Luniver press, 2010.

[160] Xin-She Yang. A new metaheuristic bat-inspired algorithm. In *Nature inspired cooperative strategies for optimization (NICSO 2010)*, pages 65–74. Springer, 2010.

[161] HM Yao, HB Vuthaluru, MO Tade, and D Djukanovic. Artificial neural network-based prediction of hydrogen content of coal in power station boilers. *Fuel*, 84(12):1535–1542, 2005.

[162] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

[163] Hongnian Zang, Shujun Zhang, and Kevin Hapeshi. A review of nature-inspired algorithms. *Journal of Bionic Engineering*, 7:S232 – S237, 2010.

[164] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.

[165] Guo-Bing Zhou, Jianxin Wu, Chen-Lin Zhang, and Zhi-Hua Zhou. Minimal gated unit for recurrent neural networks. *International Journal of Automation and Computing*, 13(3):226–234, 2016.

[166] Hao Zhou, Kefa Cen, and Jianren Fan. Modeling and optimization of the nox emission characteristics of a tangentially fired boiler with artificial neural networks. *Energy*, 29(1):167 – 183, 2004.

[167] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.